

AI Toolkit for IBM Z and LinuxONE

Lydia Parziale

Sunny Anand

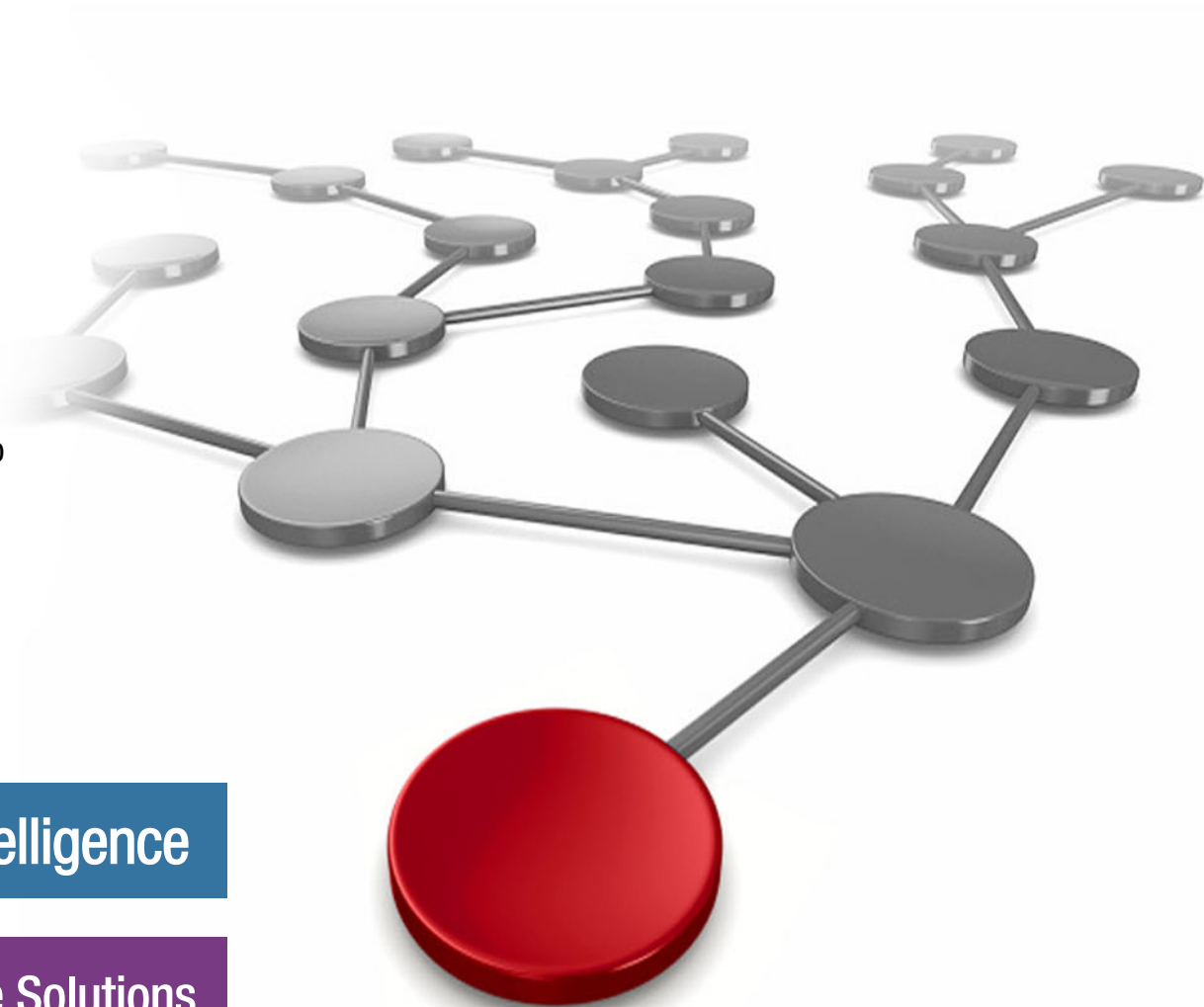
Pradipta Ghosh

Kyle Gilbertson

William Jones

KC Hema Prasanna

Prashantha Subbarao



Artificial Intelligence

Infrastructure Solutions



IBM Redbooks

AI Toolkit for IBM Z and LinuxONE

November 2025

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (November 2025)

This edition applies to the following versions of products which comprise the AI Toolkit for IBM Z and LinuxONE:

- ▶ IBM Z Deep Learning compiler 5.0.0
- ▶ IBM Z Snap ML 1.15.6
- ▶ IBM Z Accelerated for NVIDIA Triton Inference Server 1.4.1
- ▶ IBM Z Accelerated for TensorFlow 1.4
- ▶ IBM Z Accelerated Serving for TensorFlow 1.4

This document was created or updated on November 14, 2025.

© Copyright International Business Machines Corporation 2025. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
Authors	ix
Now you can become a published author, too!	xi
Comments welcome	xi
Stay connected to IBM Redbooks	xi
Chapter 1. Introduction to AI on IBM Z	1
1.1 Introduction to open-source AI	2
1.2 Significance of Open-source software	2
1.2.1 Open-source and AI	2
1.2.2 Challenges around open-source software	3
1.2.3 Open-source AI, IBM Z, and LinuxONE	3
1.3 AI accelerators on IBM Z	4
1.3.1 Telum processors for IBM z16	4
1.3.2 IBM Telum II processors for IBM z17	4
1.3.3 IBM Z Integrated Accelerator for AI: Processor features	5
1.3.4 IBM Spyre Accelerator	6
1.4 AI Toolkit for IBM Z and LinuxONE	6
1.4.1 Key Challenges Addressed by the AI Toolkit for IBM Z and LinuxONE	7
1.4.2 Benefits of the AI Toolkit for IBM Z and LinuxONE	7
1.4.3 IBM Z and LinuxONE Container Registry	7
1.4.4 Use cases	7
1.5 Components of the AI Toolkit for IBM Z and LinuxONE	8
1.5.1 AI frameworks and serving solutions	8
1.5.2 Features	9
1.5.3 Technical Support	10
Chapter 2. Overview of fraud detection use case	11
2.1 Fraud detection at scale on IBM Z	12
2.2 Card payment fraud detection and solution	12
2.2.1 Toolkit components and functions	12
2.2.2 Deployment architecture	13
2.2.3 Use case alignment	13
2.3 Credit Card Fraud use case overview	13
2.3.1 Tabular Input Data	14
2.3.2 Input Processing	14
2.3.3 Time Series Data	15
2.4 Use case model	16
Chapter 3. Introduction to PyTorch	17
3.1 PyTorch overview	18
3.2 Model training and saving	18
3.2.1 Model creation	18
3.2.2 Model training	19
3.2.3 Model saving	19
3.3 Model loading and inference	20

3.3.1	Model loading	20
3.3.2	Inference	21
3.3.3	Downloading open-Source models	21
3.3.4	Model profiling	21
3.4	Support for IBM Telum processors	22
3.5	Near real-time credit card fraud detection use case	23
3.5.1	Background information	23
3.5.2	Model construction and training	23
3.5.3	Model profiling	24
Chapter 4.	IBM Z Deep Learning Compiler	25
4.1	Introduction to ONNX	26
4.1.1	Overview of ONNX model converters and visualizers	27
4.1.2	Visualizing an ONNX model	28
4.1.3	Pytorch to ONNX model export	29
4.1.4	TorchDynamo-based ONNX Exporter	29
4.1.5	Tensorflow to ONNX model export	29
4.1.6	Recommended practices for converting ONNX models for IBM Z deployment	30
4.2	Introduction to IBM Z Deep Learning Compiler	30
4.3	Compiling a model with IBM Z Deep Learning Compiler	32
4.4	Model loading and inferencing using compiled objects	34
4.5	AI enablement on IBM Z using zDLC	35
4.6	Overview of zDLC deployment scenarios and exploiters	37
4.6.1	IBM Z Deep Learning exploiter on Linux for IBM Z	40
4.6.2	IBM Z Deep Learning exploiter on z/OS	40
4.7	Use Case: Near Real-Time Credit Card Fraud Detection	41
4.7.1	Background Information	41
4.7.2	Training and Exporting a CCFD ONNX Model using PyTorch	41
4.7.3	Building the CCFD .so using the IBM Z Deep Learning Compiler	41
4.7.4	Running the CCFD inference Python example	42
4.8	IBM Z Deep Learning Compiler Features	42
4.9	IBM Z Deep Learning Compiler Telum Support	42
4.10	IBM Z Deep Learning Compiler LLM enablement	44
4.11	IBM Z Deep Learning Compiler quantization support	45
4.11.1	Dynamic quantization by the compiler	47
4.11.2	Performance Notes	47
4.11.3	Limitations	47
4.12	IBM Z Deep Learning Compiler debug instrumentation	48
4.13	IBM zDLC performance features	48
4.13.1	Specifying input tensor dimensions	48
4.13.2	View operation targets at compile time	48
4.13.3	Device placement of operations	49
4.14	IBM Z Deep Learning Compiler scope and versioning	49
Chapter 5.	IBM Snap ML	51
5.1	Unlocking the full potential of machine learning with IBM Snap ML	52
5.1.1	Efficient inference by using IBM Integrated AI Accelerator	52
5.1.2	Model-agnostic inference pipeline	52
5.1.3	Supported model types and formats	53
5.1.4	Multi-output calibrated classifier	55
5.1.5	Multi-Threaded Inference in Snap ML	56
5.1.6	Enabling verbose logging in Snap ML	57
5.2	Graph feature preprocessor	57

5.3 C++ preprocessing pipeline	58
5.3.1 Exporting a preprocessing pipeline	58
5.3.2 Using a preprocessing pipeline in Snap ML	59
5.3.3 Understanding pipeline.json	59
5.4 Model import functionality	61
5.4.1 Import Methods	61
5.4.2 Platform Compatibility Considerations	62
5.5 Installation guide	62
5.6 Snap ML End to End Model Deployment	62
Chapter 6. IBM Z Accelerated for NVIDIA Triton Inference Server	63
6.1 Train anywhere and deploy on IBM Z by using Triton Inference Server	64
6.1.1 Triton Inference Server architecture on IBM Z	64
6.1.2 Understanding Triton Inference Server components on IBM Z	65
6.1.3 Schedulers and running models concurrently	67
6.2 Deploying strategies in production	69
6.2.1 Containerized deployment	69
6.2.2 Production readiness checklist	70
6.2.3 Model analyzer support for IBM's customized backends	70
6.3 Sample usage and deployment options	71
6.4 End-to-end model deployment use cases with IBM TIS	73
6.5 IBM Z accelerated for TIS: Best practices	73
6.5.1 Defining response cache in TIS	74
6.5.2 Using the model analyzer for performance optimization	74
6.5.3 Enabling dynamic batching in the configuration file	74
6.5.4 Using the HTTP & GRPC end points exposed	75
6.5.5 Running multiple model instances	76
6.5.6 Using logging options for better monitoring and debugging	76
Chapter 7. IBM Z Accelerated for TensorFlow	77
7.1 Introduction to TensorFlow	78
7.1.1 Models	78
7.1.2 Graph execution	78
7.1.3 Eager execution	79
7.1.4 TensorFlow functions	79
7.2 Model training and saving	80
7.2.1 Model creation	80
7.2.2 Model training	80
7.2.3 Model saving and exporting	81
7.2.4 Model loading and inference	81
7.2.5 Model profiling	82
7.3 IBM-zDNN-Plugin	83
7.3.1 Integration of IBM-zDNN-Plugin into IBM Z	83
7.3.2 Model profiling	84
7.4 Near real-time credit card fraud detection use case	85
7.4.1 Background information	86
7.4.2 Model construction and training	86
7.4.3 Model profiling	86
Chapter 8. IBM Z Accelerated Serving for TensorFlow	89
8.1 Introduction to TensorFlow Serving	90
8.1.1 Servable	90
8.1.2 ModelServer	90
8.1.3 Client APIs	91

8.1.4 Runtime.	91
8.1.5 Model deployment.	92
8.1.6 Model serving	92
8.1.7 Model Inferencing	92
8.2 Near real-time credit card fraud detection use case	93
8.2.1 Model deployment.	93
8.2.2 Model serving	94
8.2.3 Model inferences.	94
8.2.4 Model profiling.	94
Chapter 9. Other considerations	97
9.1 Security and technical considerations.	98
9.2 Licenses and support from IBM	98
Abbreviations and acronyms	99
Related publications	101
IBM Redbooks	101
Online resources	101
Help from IBM	101

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

CICS®	IBM Z®	z/OS®
Granite®	IBM z16®	z15®
IBM®	Redbooks®	z16®
IBM Spyre™	Redbooks (logo)  ®	z17™
IBM Telum®	Spyre™	
IBM Watson®	Watson Health®	

The following terms are trademarks of other companies:

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Red Hat, OpenShift are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Preface

The AI Toolkit for IBM® Z® and LinuxONE is a comprehensive suite of tools that are designed to streamline the development and deployment of AI models on IBM's enterprise-grade platforms. This toolkit empowers developers and data scientists to use the power of IBM Z and LinuxONE systems for AI workloads, offering a seamless integration with popular AI frameworks and libraries. It includes features such as automated model deployment, performance optimization, and security enhancements, making it an ideal choice for organizations seeking to harness the potential of AI in their enterprise environments.

In the rapidly evolving landscape of technology, the ability to harness the power of artificial intelligence and machine learning is paramount. This IBM Redbooks publication provides a guide to using IBM Z systems for accelerated AI workloads, offering insights into various tools and frameworks that significantly enhance performance and efficiency. As readers explore the intricacies of fraud detection and the application of deep learning models, they gain a deeper understanding of how IBM Z serves as a game-changer in this domain.

This IBM Redbooks publication discusses the architecture, components, and deployment strategies of the AI Toolkit for IBM Z and LinuxONE. The publication explores how this toolkit integrates with open-source frameworks such as TensorFlow, PyTorch, Snap ML, and ONNX, and how it uses IBM Z hardware accelerators like the IBM Telum and IBM Spyre processors to deliver high-performance AI inferencing.

This publication also provides practical guidance through real-world use cases, including near real-time credit card fraud detection, to demonstrate how AI models can be trained, optimized, and deployed efficiently on IBM Z. Readers gain insights into model conversion, quantization, performance tuning, and containerized deployment using IBM customized backends for Triton Inference Server and TensorFlow Serving.

Whether you are a data scientist, AI engineer, system architect, or IT decision-maker, this Redpaper helps you build and scale AI solutions on IBM Z and LinuxONE platforms with confidence, security, and enterprise-grade support.

Authors

This paper was produced by a team of specialists from around the world working at IBM Redbooks Center.

Lydia Parziale is a Project Leader for the IBM® Redbooks® team in Poughkeepsie, New York, US, with domestic and international experience in technology management, including software development, project leadership, and strategic planning. Her areas of expertise include business development and database management technologies. Lydia is a certified PMP and an IBM Certified IT Specialist with an MBA in Technology Management and has been employed by IBM for over 25 years in various technology areas.

Sunny Anand is the Product Owner and Team Lead for the IBM Z Deep Learning Compiler product. Sunny is currently based out of the IBM Austin Office. He has worked in areas of Machine Learning and AI infrastructure for the last 10 years at IBM. He has worked on products like IBM Watson® Studio, IBM Watson Health® Data Lake, IBM CloudPak for Data(Platform Orchestration), and IBM Z® Deep Compiler. He is well known for his deep

expertise in DataOps and MLOps and has published papers in these fields. He has also published work in Machine Learning on various learning platforms.

Pradipta Ghosh is the Architect for AI on IBM Z. He is responsible for the design and development of high performant AI inference Solution for IBM Z. He has been with IBM India Systems Development Lab and has over 16 years of experience with Inference Solutions, AI framework, File system, Device Driver across different platforms. As a significant open-source contributor, Pradipta has contributed to wide range of AI frameworks and libraries i.e NumPy, XGboost, SciPy, Triton Inference Server etc. He has also co-authored papers and posters in prestigious external conferences i.e SciPy 2020, IEEE conferences etc.

Kyle Gilbertson is a developer on IBM Z and IBM LinuxONE for TensorFlow, TensorFlow Serving, and PyTorch, as well as the security focal for AI Toolkit for IBM Z and IBM LinuxOne. He has multiple years of experience in software development field and has worked at IBM for 3 years. His areas of expertise include C++, Machine Learning, and Deep Neural Network frameworks.

William Jones is a developer on IBM Z and IBM LinuxONE for TensorFlow, TensorFlow Serving, and PyTorch, as well as the security focal for AI Toolkit for IBM Z and IBM LinuxOne. He has multiple years of experience in software development field and has worked at IBM for 3 years. His areas of expertise include C++, Machine Learning, and Deep Neural Network frameworks.

KC Hema Prasanna is an AI Engineer based in India with over eight years of experience in software development, specialising in IBM Z and mainframe technologies. Her academic background combines Engineering at the undergraduate level with a Master's in Business Administration. For the past two years at IBM, Hema has been a key contributor to the IBM Z Accelerated for NVIDIA Triton Inference Delivery Team. She has played an integral role across all phases of the product lifecycle, including backend design and development, bug resolution, and end-to-end integration. Hema possesses deep expertise in IBM Z Accelerated for NVIDIA Triton Inference, with a strong understanding of its features, capabilities, and usage patterns—particularly in the context of enabling AI within enterprise infrastructure.

Prashantha Subbarao is the Product Owner and team lead for AI Toolkit for IBM Z and LinuxONE as well as AI Frameworks such as PyTorch and TensorFlow on IBM Z. He has around 25 years of experience in IBM. He experienced in development and transformation projects across IBM Z and Power systems. His areas of expertise include Acceleration for AI Frameworks and Serving solutions, Data Science, and performance computing.

Thanks to the following people for their contributions to this project:

Kelly Yang, Mahalakshmi Pathivada
IBM

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an email to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, IBM Redbooks
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on LinkedIn:

<https://www.linkedin.com/groups/2130806>

- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/subscribe>

- ▶ Stay current on recent Redbooks publications with RSS Feeds:

<https://www.redbooks.ibm.com/rss.html>



Introduction to AI on IBM Z

This chapter begins with open-source AI and its impact on the evolution and growth of artificial intelligence (AI). It examines artificial intelligence and machine learning, and explores the significance of AI on IBM Z.

It also describes the software and hardware optimizations that enable accelerated real-time inference on IBM Z. The chapter introduces the AI Toolkit for IBM Z and LinuxONE, and explains how it provides a robust ecosystem of open-source AI solutions for accelerated inference in supported, secure-ready environments.

The goal is to provide the essential knowledge needed to understand the symbiotic relationship between AI, open-source software, security, and acceleration with IBM Z, setting the stage for deeper explorations in this IBM Redbooks publication.

This chapter includes the following sections:

- ▶ 1.1, “Introduction to open-source AI” on page 2
- ▶ 1.2, “Significance of Open-source software” on page 2
- ▶ 1.3, “AI accelerators on IBM Z” on page 4
- ▶ 1.4, “AI Toolkit for IBM Z and LinuxONE” on page 6
- ▶ 1.5, “Components of the AI Toolkit for IBM Z and LinuxONE” on page 8

1.1 Introduction to open-source AI

Open-source software impacted artificial intelligence (AI) and plays a key role in its evolution. Greater accessibility, rapid iteration, and increased collaboration among developers, data scientists, researchers, and the broader AI community have transformed AI and accelerated its evolution and maturity.

1.2 Significance of Open-source software

Open-source software is developed and maintained through open collaboration. It is available for anyone to use, examine, alter, and redistribute, typically at no cost. The term “open source” generally refers to a community-based approach to creating intellectual property, such as software, through open collaboration, inclusiveness, transparency, and frequent public updates.

Open-source software now plays a vital role in computing, with open-source technologies providing the foundation of the Internet, business computing, and personal computing. Virtually all computing devices now contain open-source code, typically adopted by developers to perform fundamental operations and, in many cases, more advanced functions.

Open source has become mainstream and gained immense popularity in recent years. A 2021 O'Reilly survey on open source usage, commissioned by IBM¹, had the following responses:

- ▶ 94% of the respondents rated open source as better than or equal to proprietary software.
- ▶ 55% of the organizations use 5 or more types of open-source software.
- ▶ 77% of the surveyed enterprises agreed that they meet their objectives while minimizing costs by using open-source software.

1.2.1 Open-source and AI

Open source has emerged as a powerful driver of innovation and accessibility across multiple fields. It has had a significant impact on artificial intelligence (AI), playing a key role in its evolution.

Open-source AI offers several benefits:

- ▶ Accelerated innovation
- ▶ Cost efficiency
- ▶ Community support
- ▶ Increased transparency
- ▶ Freedom to modify and share code
- ▶ Wider collaboration by making advanced AI technology more accessible to a broader range of users and organizations

Open access to code encourages experimentation and the development of new AI techniques and applications. Open-source AI tools are often freely available, reducing financial barriers to entry for businesses and individuals.

¹ <https://www.itprotoday.com/ibm-cloud/survey-open-source-cloud-technologies-fit-devs-like-a-glove>

Because the source code is publicly available, it can be thoroughly reviewed and scrutinized, leading to greater trust in AI systems and accountability for their functionality. Access to the source code allows users to modify and adapt open-source AI models to meet specific needs and applications. A large community of developers can contribute to the development and improvement of AI models, accelerating innovation and producing better solutions.

Artificial intelligence is one of the most powerful technological forces shaping the world today. From assisting health-care professionals in making more informed decisions to improving the efficiency of power grids and facilitating scientific research, AI has the potential to bring substantial changes to many aspects of life.

1.2.2 Challenges around open-source software

Open-source AI is reshaping enterprise innovation, offering tools that enable smarter decision-making and improved operational efficiency. Enterprises benefit from this rapid innovation cycle, gaining access to cutting-edge technologies and improvements without the lengthy development times often associated with proprietary software. However, unmaintained and unverified open-source software can introduce significant security and compliance risks.

Security concerns and inadequate support remain key reasons for the nonadoption of open-source software in the United States. Open-source AI software can be vulnerable to security risks, including code injection, data breaches, and other malicious activities. The public nature of the code can make it easier for malicious actors to identify and exploit vulnerabilities. In addition, quality control issues, compatibility challenges, and intellectual property concerns are major obstacles enterprises face when adopting open-source software.

1.2.3 Open-source AI, IBM Z, and LinuxONE

IBM Z and LinuxONE feature state-of-the-art hardware and software capabilities designed to optimize AI at scale alongside enterprise-critical workloads and data.

Mainframes host mission-critical applications and high-volume transaction processing systems for some of the world's largest enterprises, including major banks, insurance and financial services providers, airlines, manufacturing companies, and healthcare organizations. The performance, reliability, and security advantages of IBM Z mainframes ensure that they remain the primary platform for these core applications.

With vast amounts of mission-critical data in their systems, mainframe customers often prefer to keep AI inferencing on the platform rather than outsource it to a separate system. Several factors drive accelerated AI inference on the platform:

- Data locality

Keeping AI processing on the mainframe ensures that data remains onsite, eliminating the need to transfer it to external systems such as cloud services. This reduces network bottlenecks and enhances performance by minimizing data transfer times.

- Data privacy

Many mainframe customers, especially those in finance and healthcare, handle sensitive data on the platform. Processing AI tasks on IBM Z helps maintain compliance with data protection regulations because the data does not leave the highly secure mainframe environment.

- ▶ Real-time processing

Mainframes handle massive amounts of data with high transaction volumes and exceptional reliability. Integrating AI directly on the mainframe enables real-time data analysis and decision-making without the latency and security risks that are associated with moving data to external AI processing units.

- ▶ Resource usage and cost efficiency

By using existing mainframe infrastructure for AI tasks, businesses can avoid the additional cost of setting up separate AI processing systems.

1.3 AI accelerators on IBM Z

This section provides an overview of the current state of IBM Z hardware capability for AI processing, including integrated AI accelerators and PCIe attached accelerators available on IBM z16® and IBM z17™ systems. The following hardware components are included:

- ▶ Telum on-chip AI accelerators on IBM z16 machines
- ▶ Telum II processors on IBM z17 machines
- ▶ IBM Spyre™ Accelerator on IBM z17 machines (available 4Q 2025)

1.3.1 Telum processors for IBM z16

The IBM Telum® processor is an advanced microprocessor that powers many of the features and benefits of the IBM z16®. It is designed to help customers achieve business insights at scale across banking, finance, trading, insurance applications, and customer interactions.

The Telum processor is a 7 nm microprocessor designed to deliver AI-driven insights without sacrificing response time in high-volume transactional workloads. It features eight cores, each clocked at over 5 GHz and supported by a 32 MB private Level-2 (L2) cache. These combine to form a 256 MB virtual Level-3 (L3) cache and a 2 GB Level-4 (L4) cache. With 1.5 times more cache per core than its predecessor, Telum boosts per-thread performance and overall capacity, ensuring rapid response times for complex transactions, especially when enhanced by real-time AI inference.

In addition to its traditional processing cores (CPUs), Telum adds AI accelerator cores to each chip. These AI accelerator cores are designed for running deep learning workflows with over six teraflops (TFLOPs) of compute capacity per chip. With two sets of cores, Telum is specifically designed to run AI workflows alongside batch and other processing workloads.

1.3.2 IBM Telum II processors for IBM z17

The IBM Telum II processor (shown in Figure 1-1 on page 5) is the next-generation IBM Z processor. It offers more advanced AI features, built-in Data Processing Unit (DPU) capabilities, enhanced acceleration, and significantly larger cached memory. It is designed to scale processing capacity across next-generation IBM Z mainframe systems, accelerating the use of both traditional AI models and large language models (LLMs) through a new ensemble AI method.

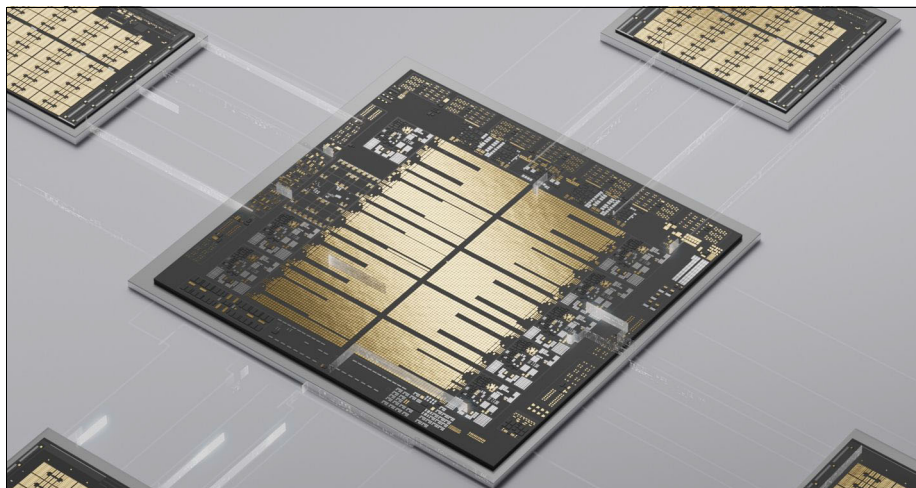


Figure 1-1 IBM Telum II processor

The Telum II processor includes a built-in low-latency DPU for accelerated I/O. The main processor contains four interconnected core clusters, each with eight programmable micro-controller cores, ensuring cache coherency among the 32 cores. The chip features ten 36 MB L2 caches: one per core, one for the DPU, and one serving as the overall chip cache capacity.

In addition, Telum II provides a larger pool of virtual L3 and L4 cache, each increased by 40% compared to the previous generation, to 360 MB and 2.8 GB respectively. These enhancements deliver meaningful performance improvements for AI workloads and high-volume transactional processing.

1.3.3 IBM Z Integrated Accelerator for AI: Processor features

The Telum processor includes on-chip AI acceleration, enabling near real-time analytics and decision-making directly on the processor without requiring additional hardware. This capability allows businesses to gain immediate insights and respond proactively to emerging trends and issues. Telum II enhances AI acceleration and inferencing with four times the compute power of Telum and coherently connected processing clusters.

High availability

The Telum processor's reliability and high availability help ensure continuous operation with minimal downtime, which is essential for mission-critical applications. Telum II offers redundant connections to processors, enabling interruption-free service by maintaining operations through attached I/O via an alternative path while a processor undergoes repair.

Low latency

The Telum processor architecture is optimized for low latency, which is critical for high-speed transaction processing and real-time analytics. Telum II increases overall available cache by 40%, with ten L2 caches and larger L2, L3, and L4 capacities, improving off-chip bandwidth and latency.

High processing power

The Telum processor is designed for high-performance computing, enabling IBM z16 to handle large-scale transactions and analytics efficiently. Telum II is projected to operate at 5.5 GHz, offering improved speed while maintaining efficient management of large-scale

transactions. IBM z17 enables businesses to process 50% more AI inference operations per day than IBM z16.

Advanced security

The Telum processor supports advanced encryption techniques, including quantum-safe encryption, to protect sensitive data against current and future cyberthreats. Telum II enhances security for A and Z bus links, strengthening data protection across mainframe interconnects and supporting secure, reliable operations for mission-critical workloads and helping to ensure compliance with regulatory standards.

Note: A and Z bus links are high-speed connections between processors and system components in IBM mainframes, ensuring fast and secure data transfer.

Energy efficiency

The Telum processor design optimizes power consumption, contributing to the energy efficiency of the IBM z16 and reducing operational costs. Telum II improves energy efficiency further, with up to 15% core power reduction based on centralized I/O and DPU design.

1.3.4 IBM Spyre Accelerator

The IBM Spyre Accelerator is designed to provide additional AI compute capabilities that complement the Telum II processor on IBM z17. Together, they create optimized environments to support multi-model AI methods. The IBM Spyre Accelerator is engineered to bring generative AI capabilities to the mainframe, including running AI assistants and using enterprise data contained within the system.

Built on a 5 nm process, the IBM Spyre Accelerator expands the AI computing capabilities of Telum II processors. Each card contains 32 AI-focused accelerator cores, purpose-built to support demanding tasks such as machine learning, large language models (LLMs), and other AI-driven workloads. By using low-precision numeric formats such as int4 and int8, Spyre cores improve energy efficiency and reduce memory usage when running complex AI models.

These accelerator cards connect via PCIe to the system's I/O subsystem, allowing multiple cards to be clustered and scaled across IBM Z systems. Each additional Spyre card adds 32 more accelerator cores to the system's AI resources, in addition to the 32 AI cores already embedded within the Telum II processor.

1.4 AI Toolkit for IBM Z and LinuxONE

The AI Toolkit for IBM Z and LinuxONE is a family of popular open-source AI frameworks with IBM Elite support (as part of [IBM Selected Support](#)), adapted for IBM Z and LinuxONE hardware. With a curated suite of AI software, clients can manage AI model life cycles in one place, enabling quick deployment across a wide range of use cases.

This select set of containerized software has been evaluated by IBM. Pre-built images are available through the IBM Z and LinuxONE Container Registry.

These images run in Linux on Z environments, including IBM z/OS® Container Extensions. The software is optimized for IBM Z and LinuxONE platforms, with the ability to use the Integrated Accelerator for AI.

1.4.1 Key Challenges Addressed by the AI Toolkit for IBM Z and LinuxONE

Using open-source AI on IBM Z offers benefits such as increased accessibility, accelerated performance through the Telum on-chip accelerator, and streamlined development with pre-built tools and libraries. However, concerns about security and lack of support remain significant obstacles to adoption.

Many enterprises that rely on IBM Z and LinuxONE are embracing open-source software to modernize application development and make use of new technologies such as AI. These organizations have developed substantial skills in open-source AI frameworks that can be applied across hybrid environments. However, unsupported open-source software can introduce security and compliance risks, which is especially concerning for sensitive production environments.

The AI Toolkit for IBM Z and LinuxONE provides a supported, high-performing, and validated environment that helps mitigate challenges related to open-source support for enterprise AI solutions.

1.4.2 Benefits of the AI Toolkit for IBM Z and LinuxONE

The AI Toolkit for IBM Z and LinuxONE is designed to deliver high-performance runtimes and serving environments that fully use the advanced software and hardware capabilities of the IBM Z platform. These tools are built for ease of use, offering best-in-class AI solutions in an accessible, production-ready form. They are seamlessly integrated with the IBM Z Integrated Accelerator for AI, enabling significantly faster inference for demanding workloads.

In addition, the AI Toolkit provides enterprise-grade support and simplifies the deployment of open-source AI solutions on a robust, enterprise-scale system.

Beyond service and support, the open-source AI libraries undergo an extensive secure engineering process. The source code, libraries, and packages are scanned, vetted, and continuously monitored for vulnerabilities. Together, these capabilities make the AI Toolkit for IBM Z and LinuxONE both high performing and trustworthy.

1.4.3 IBM Z and LinuxONE Container Registry

The IBM Z and LinuxONE Container Image Registry includes open-source software packaged in container images that is often used as the foundation for new composite workloads. It provides a secure and trustworthy source of content. Components of the AI Toolkit for IBM Z and LinuxONE are freely available through the IBM Z and LinuxONE Container Image Registry.

1.4.4 Use cases

A wide range of use cases can be implemented with AI software components for the IBM hardware platforms:

- ▶ **Claims fraud detection:** A state government in the United States determined that its process for identifying fraudulent claims was manual and time-intensive, taking up to 40 hours per case. IBM demonstrated that by using IBM z16 with the Integrated Accelerator for AI, adding transactional fraud detection for OLTP transactions resulted in only 2 milliseconds of additional response time.
- ▶ **Clearing and settlement risk analysis:** A payment card processor explored the use of AI to identify trades or transactions with high-risk exposure before settlement, reducing liability,

charge-backs, and costly investigations. IBM validated that the IBM z16 is designed to score business transactions at scale, delivering the capacity to process up to 300 billion deep inferencing requests per day with 1 millisecond of latency.

- ▶ Anti-money laundering (AML) screening: A European bank needed to introduce AML screening into its instant payments operational flow. End-of-day AML screening was no longer sufficient due to stricter regulations. IBM demonstrated that the IBM z16 with the Integrated Accelerator for AI provides four times faster response time compared with the IBM z15® when both systems run equivalent OLTP workloads with batched fraud detection.

1.5 Components of the AI Toolkit for IBM Z and LinuxONE

The AI Toolkit for IBM Z and IBM LinuxONE offers IBM Elite support (as part of [IBM Selected Support](#)) for popular open-source and IBM nonwarranted AI programs. This support gives organizations confidence in deploying open-source AI in production, including the following programs:

- ▶ PyTorch
- ▶ TensorFlow
- ▶ TensorFlow Serving
- ▶ NVIDIA Triton Inference Server
- ▶ IBM Snap ML
- ▶ IBM Z Deep Learning Compiler

The offering simplifies support licensing and provides IBM Certified containers for popular open-source and IBM nonwarranted AI programs. These containers are optimized to run seamlessly on IBM Z and IBM LinuxONE. They are tested for vulnerability and security by IBM, enabling organizations to deploy open-source AI with confidence.

1.5.1 AI frameworks and serving solutions

The frameworks that make up the AI Toolkit for IBM Z and LinuxONE and deployment options are shown in Figure 1-2 on page 9. The following list provides a brief overview of each of these frameworks and deployment options:

- ▶ [IBM Z Accelerated Serving for TensorFlow](#): Flexible and high-performing serving platform for ML/DL models optimized to use IBM Z Integrated Accelerator for AI.
- ▶ [IBM Z Accelerated for TensorFlow](#): Popular Machine Learning and Deep Learning (ML/DL) lifecycle management platform optimized to run on Z and use IBM Z Integrated Accelerator for AI.
- ▶ [IBM Z Accelerated for SnapML](#): A library that optimizes the training/scoring of popular ML models to use IBM Z Integrated Accelerator for AI.
- ▶ [IBM Z Deep Learning Compiler](#): Generates a program from any Open Neural Network Exchange (ONNX) DL models to run on z/OS or Linux on Z optimized to use IBM Z Integrated Accelerator for AI.
- ▶ [IBM Z Accelerated for NVIDIA Triton Inference Server](#): High-performance inference server that supports the deployment of ML or DL models at scale optimized to use IBM Z Integrated Accelerator for AI.
- ▶ [IBM Z Accelerated for PyTorch](#): A popular Machine Learning framework based on Torch library, which is used for applications such as language processing and computer vision and optimized to run on IBM Z and use IBM Z Integrated Accelerator for AI.

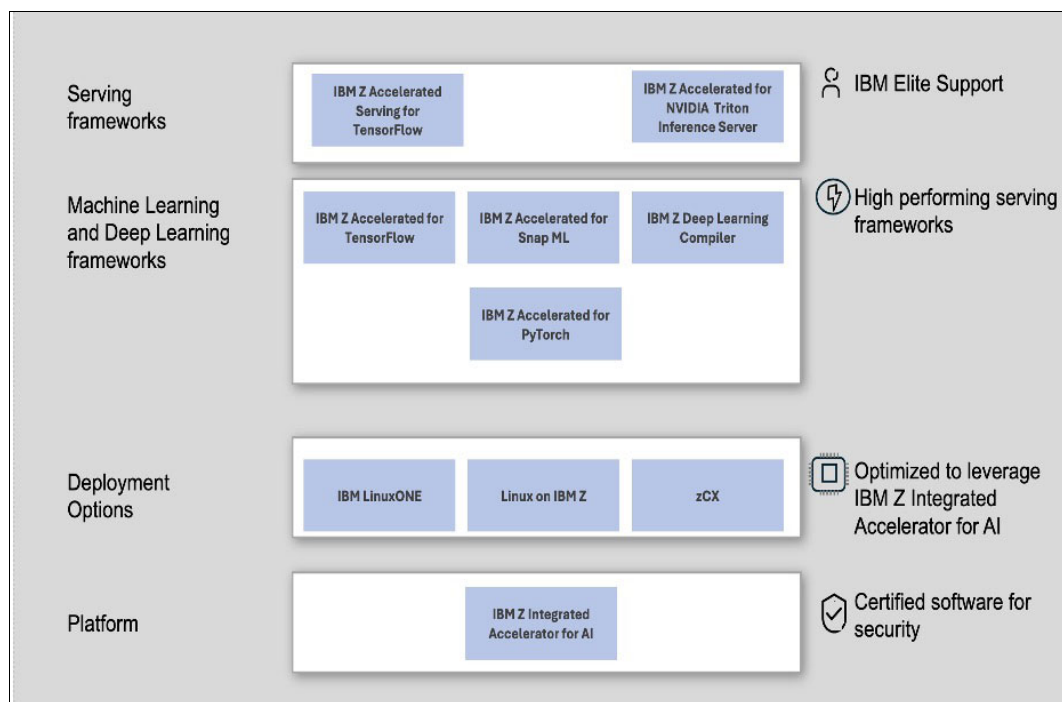


Figure 1-2 Frameworks and deployment options in AI toolkit for IBM Z and LinuxONE

For more information, see the IBM Redbooks publication *AI for Linux on IBM Z and LinuxONE applications and examples*, REDP-5756.

1.5.2 Features

By using hardware accelerators, compiler optimizations, and containerized environments, developers, and data scientists can efficiently deploy and scale machine learning models while minimizing infrastructure workloads. The following features help maximize the performance of AI workloads on IBM Z and LinuxONE:

- ▶ **Free download of AI frameworks**
Reduce costs and complexity and accelerate time to market with lightweight and free-to-download tools and runtime packages.
- ▶ **TensorFlow compatibility**
Accelerate seamless integration of TensorFlow with IBM Z Accelerated for TensorFlow to develop and deploy machine learning (ML) models on neural networks.
- ▶ **AI frameworks integration**
Use IBM Z Accelerated for NVIDIA Triton Inference Server to streamline and standardize AI inferences by deploying ML or DL models from any framework on any GPU-based or CPU-based infrastructure.
- ▶ **ML models with TensorFlow Serving**
Harness the benefits of TensorFlow Serving, a flexible and high-performance service system, with IBM Z Accelerated for TensorFlow Serving to help deploy ML models in production.

- ▶ Compile models with IBM Z Deep Learning Compiler
Convert ML models into a code that can be run on z/OS or LinuxONE with the help of IBM Z Deep Learning Compiler (IBM zDLC).
- ▶ Run Snap ML
Use IBM Z Accelerated for Snap ML to build and deploy ML models with Snap ML, an IBM nonwarranted program that optimizes the training and scoring of popular ML models.
- ▶ PyTorch compatibility
Accelerate seamless integration of PyTorch with IBM Z Accelerated for PyTorch to develop and deploy machine learning (ML) models on neural networks.

1.5.3 Technical Support

Although open-source software helps make AI more accessible, affordable, and innovative, the successful implementation of these frameworks requires the correct level of support. With the introduction of the AI Toolkit for IBM Z and IBM LinuxONE, organizations can use a support offering to deploy and accelerate the adoption of popular open-source AI frameworks on IBM z/OS and IBM LinuxONE platforms.

The AI Toolkit consists of IBM Elite Support and IBM Secure Engineering, which vet and scan open-source AI serving frameworks and IBM Certified containers for security vulnerabilities and validate compliance with industry regulations.

For more information about obtaining support and information about the offerings that are supported within the AI Toolkit for IBM Z and IBM LinuxONE, see [AI Toolkit for Z and LinuxONE](#). Details about the level of support are available in the [What to Expect From Support](#) document.



Overview of fraud detection use case

In the digital age, financial fraud is an increasingly prevalent threat. With the rise of online transactions and digital banking, fraudsters are finding new ways to exploit vulnerabilities in systems. This makes fraud detection more critical than ever before. Artificial Intelligence (AI), with its ability to analyze large volumes of data and identify complex patterns, can be a powerful tool in combating fraud. This chapter describes the Fraud Detection use case on IBM Z in detail and the following chapters explore how AI Toolkit for IBM Z and LinuxONE can be used to solve the problem of Fraud Detection and support real-time inference on IBM Z systems.

This chapter includes the following topics:

- ▶ 2.1, “Fraud detection at scale on IBM Z” on page 12
- ▶ 2.2, “Card payment fraud detection and solution” on page 12
- ▶ 2.3, “Credit Card Fraud use case overview” on page 13
- ▶ 2.4, “Use case model” on page 16

2.1 Fraud detection at scale on IBM Z

To counter increasingly sophisticated and technology-enabled fraud that targets the financial services industry, financial institutions and their technology partners are developing increasingly advanced solutions to detect and prevent fraudulent activity. These solutions include techniques that use multiple predictive deep learning and generative AI models. However, advanced anti-fraud models typically run on peripheral hardware separated from the transactions and data, creating challenges in deploying AI models at scale for large institutions that run mission-critical transactions on mainframe computers. The need for additional processing power and technology resources continues to grow as the models become larger and more complex.

IBM addressed this barrier with the release of its Telum processor for the IBM z16 mainframe in 2022, which incorporates an AI accelerator to support AI inferencing directly in the mainframe environment. This breakthrough enabled large banks to run complex anti-fraud models against all transactions in real time without the latency and throughput issues that can occur when core transactions are sent to peripheral fraud systems.

IBM introduced second-generation AI accelerators as part of its z17 system. These accelerators are significantly more powerful, providing sufficient compute capacity to run multiple AI models simultaneously. They are also tuned to run complex AI workloads, such as fraud detection, in real time directly in the mainframe environment.

2.2 Card payment fraud detection and solution

Fraudulent credit card transactions are a major concern for credit card companies. Detecting potential fraudulent transactions in real time can help prevent customers from being charged for unauthorized activity. The goal is to build a model that can determine whether transactions are fraudulent in real time.

Payment fraud is defined as intentional deception or misrepresentation that results in an unauthorized benefit. Fraud schemes are becoming increasingly complex and difficult to identify. Industries lose billions of dollars annually because of fraud.

The ideal solution is to prevent fraudulent payments without slowing down legitimate transactions. Achieving this requires the adoption of a comprehensive fraud business architecture that applies predictive analytics. Technological advancements can enable a shift from post-payment to pre-payment fraud detection. The AI Toolkit for IBM Z and LinuxONE supports these advancements while also providing traditional mainframe benefits.

To counter payment fraud effectively, financial institutions require robust, real-time AI models that are scalable, flexible, and enterprise-ready. The AI Toolkit for IBM Z and LinuxONE delivers this capability by providing optimized support for leading open-source AI frameworks alongside enterprise-grade deployment tools.

The AI Toolkit for IBM Z and LinuxONE provides IBM Elite Support for several popular open-source AI frameworks and serving solutions.

2.2.1 Toolkit components and functions

The toolkit includes containerized distributions of six open-source AI and machine learning tools:

- ▶ TensorFlow, PyTorch
- ▶ TensorFlow Serving
- ▶ IBM Z Deep Learning Compiler (zDLC)\
- ▶ Snap ML, and Triton Inference Server

These are available through the IBM Container Registry (ICR). Each component is tailored for seamless integration within enterprise environments and delivered in a format optimized for IBM Z and LinuxONE platforms.

2.2.2 Deployment architecture

On IBM Z systems, these AI solutions function similarly to how they operate in x86 environments, but the solutions are optimized to use the advantages of the s390x architecture. Enhancements include support for vector processing and integrated on-chip AI acceleration powered by the Telum processor. These optimizations significantly reduce inference latency for complex fraud detection models running in production.

2.2.3 Use case alignment

Although the toolkit supports a wide variety of machine learning (ML) and deep learning (DL) tasks, it is especially well tuned for high-throughput, low-latency scenarios such as credit card fraud detection. Use cases of this type use Telum on-chip acceleration and deploy on IBM Z systems for real-time inference with transactional data.

Running AI inference directly within the transaction processing pipeline enables monitoring of potentially fraudulent transactions in-stream, avoiding round-trips to external inference engines.

There are several ways to train and deploy a credit card fraud model on IBM Z, and each component of the AI Toolkit can support this in a unique way. The following sections explore each component of the AI Toolkit in detail and explain how it can help implement the credit card fraud model on IBM Z systems.

2.3 Credit Card Fraud use case overview

This section provides a high-level overview of the data and the model for the credit card fraud use case. Subsequent chapters discuss the detailed implementation of the model with various components of the AI Toolkit. For more information, see the sample open source credit card transaction dataset, which is published in the following locations:

- ▶ [TabFormer](#)
- ▶ [Kaggle](#).

2.3.1 Tabular Input Data

Table 2-1 shows sample input data from a comma-separated values (CSV) file of customer transactions with 15 rows.

Table 2-1 Input data sample

Column name	Data type	Example
User	Integer	19
Card	Integer	1
Year	Integer	2025
Month	Integer	1
Day	Integer	26
Time	String	06:21
Amount	String	\$134.09
Use Chip	String	Swipe Transaction
Merchant Name	Integer	3527213246127876953
Merchant City	String	La Verne
Merchant State	String	CA
Zip	Float	91750.0
MCC	Integer	5300
Errors?	String	NaN
Is Fraud?	String	No

2.3.2 Input Processing

During the initial training run, categorical and string-based features are one-hot encoded, and the encoders are fit to the training dataset. These fitted encoders are then persisted and reused for subsequent training and inference sessions, ensuring consistency in data transformation throughout the model's lifecycle.

A total of 221 features are generated from the original columns in the dataset, with the "Is Fraud?" column designated as the label and the remaining 220 columns serving as input features.

Decision trees and random forests

Decision trees and random forests can be used with the one-hot encoded integer inputs unchanged.

Neural networks

Neural networks can be used by treating the one-hot encoded integer inputs as floating-point values and feeding them into a model that includes a fully connected layer. The number of neurons in the hidden layer of the fully connected layer is determined when the model is constructed, which can potentially allow more connections to be recognized than in a decision tree or random forest.

Efficient preprocessing is critical for enabling accurate and consistent model inference at scale. In this example, the credit card transaction data undergoes transformation and encoding to prepare it for training and inference with machine learning models, as shown in Table 2-2 on page 15.

Table 2-2 Encoded values

Source Column Name	Encoded Column Name	Number of Columns
User	User	1
Card	Card	1
Year	Year_Month_Day_Time	1
Month		
Day		
Time		
Amount	Amount	1
Use Chip	Use Chip	3
Merchant Name	Merchant Name	50
Merchant City	Merchant City	42
Merchant State	Merchant State	25
Zip	Zip	50
MCC	MCC	24
Errors?	Errors?	24
Is Fraud?	Is Fraud?	1

2.3.3 Time Series Data

With the ability to detect fraud for a given sample, multiple input samples can be passed to certain models so that information can flow between transactions and incorporating temporal context.

Recurrent Neural Networks (RNNs)

These models are similar to neural networks, but they process sequences of input samples. The number of samples in each sequence is set when the model is constructed.

There are two common implementations of RNNs:

1. GRU (Gated Recurrent Unit): A simplified version of an RNN that uses gates to control the flow of information, balancing memory retention and forgetting. It is computationally efficient and effective for many sequence tasks.
2. LSTM (Long Short-Term Memory): An advanced RNN variant that uses a more complex structure with multiple gates (input, forget, and output) to better handle long-term dependencies in sequences, making it robust for tasks requiring extended context.

Like a fully connected (dense) layer with a hidden layer and hidden weights, GRU and LSTM models maintain internal states that serve a similar function but for the entire sequence of inputs.

During training, each sample in the sequence is processed in a loop, updating the same state after each sample. This allows the state to depend on all samples in the sequence, not just a single sample.

Samples in the sequence can be processed in forward order, reverse order, or in both forward and then reverse order. The output can include all samples in the sequence or only the last sample, and the state can also be output. The outputs can represent the following types of information:

- ▶ Information about each sequence
- ▶ Information about the last sample in the sequence
- ▶ Holistic information about the entire sequence
- ▶ State information passed across RNN layers

2.4 Use case model

The TensorFlow and PyTorch models are RNN-based, implemented using either GRU or LSTM. They contain two consecutive RNN layers that are processed in forward order. The first RNN layer returns all sequences, which are then passed as inputs to the second RNN layer, similar to a fully connected layer. The output of the second RNN layer is passed to a dense layer for classification.

The final output indicates whether the last sample in the sequence is fraudulent.



Introduction to PyTorch

This chapter explores IBM Z Accelerated for PyTorch, which comprises PyTorch and IBM enhancements that were developed to use the capabilities of IBM Z. The chapter introduces PyTorch, including creating, training, saving, and loading models, and examines the optimizations available for inferencing. Also, the chapter presents a use case that creates a model for near real-time credit card fraud detection using PyTorch and performs optimized inference calls that use IBM Z capabilities.

This chapter includes the following topics:

- ▶ 3.1, “PyTorch overview” on page 18
- ▶ 3.2, “Model training and saving” on page 18
- ▶ 3.3, “Model loading and inference” on page 20
- ▶ 3.4, “Support for IBM Telum processors” on page 22
- ▶ 3.5, “Near real-time credit card fraud detection use case” on page 23

3.1 PyTorch overview

PyTorch is an industry-leading, open-source framework for training and performing inference on deep learning models. It was originally developed by Meta as a Python port of the Torch Lua library and has been part of the Linux Foundation since 2022.

In PyTorch, you create models by writing a Python program that specifies the layers and tensor-based operations of the deep neural network and compiling them into a model. PyTorch automatically computes the gradients for the operations in your model, which simplifies the training process. You can persist the model to disk and then load it in another program that performs inference using inputs from your applications. PyTorch also has a C++ interface. Its focus on simplicity, low CPU usage, efficient memory management, and a robust community and ecosystem make it a strong choice for AI applications of all sizes.

PyTorch integrates well with other common Python data science and machine learning frameworks such as NumPy, SciPy, and Scikit-learn. You can extend PyTorch through Python code or C++-based APIs. PyTorch also supports popular AI accelerators to speed up computation on other platforms. Later sections discuss acceleration options for IBM Z servers.

Many resources are available for learning how to use PyTorch including the following resources:

- ▶ [Welcome to PyTorch Tutorials](#)
- ▶ [PyTorch documentation](#).

3.2 Model training and saving

This is a brief overview of training a model from scratch in PyTorch. For a more thorough explanation of this code, see [PyTorch Quickstart](#).

3.2.1 Model creation

Creating and training models from scratch entails creating a Python class that extends `torch.nn.Module` and defining the architecture of the model in the `__init__` method and defining a method for the forward pass that starts the model (see Example 3-1).

Example 3-1 Extending torch.nn.module

```
class NeuralNetwork(nn.Module):
    def __init__(__self):
        super().__init__()
        self.flatten = nn.Flatten()
        # This is the architecture of the neural network
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10) )
        def forward(self, x):
            x = self.flatten(x)
            logits = self.linear_relu_stack(x)
```



```
        return logits

model = NeuralNetwork()
print(model)
```

3.2.2 Model training

To train a model, choose a loss function and an optimizer. Selecting the appropriate loss function and optimizer for your application is outside the scope of this IBM Redbooks publication. Include boilerplate code to implement the training loop. A sample of training a model in PyTorch is shown in Example 3-2.

Example 3-2 Sample of training a model in PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}    [{current:>5d}/{size:>5d}]")

epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-----") train(train_dataloader,
    model, loss_fn, optimizer)
    # For brevity, the test part of this example has been omitted.
    # Refer to the PyTorch tutorial for a complete code listing.

print("Done!")
```

3.2.3 Model saving

PyTorch saves the learned parameters in a data structure called the `state_dict`. `torch.save` and these can be saved to a file so they can be used again later without having to retrain the model. There the `state_dict` is saved to a file named `model.pth`. The following example is a command that is used to save to a file:

```
torch.save(model.state_dict(), "model.pth")
```

To use these weights, ensure that the program has access to the `torch.nn.Module` subclass defined in Example 3-1 on page 18. These definitions must match. Move the code to a reusable Python module, copy and paste it, or use other Python facilities to save the model class to a file and reload it. If you use a third-party module, locate the matching source code or documentation.

Instead, the **`torch.save`** command can save the model weights and class definition together in the same file. However, this method uses the Python pickle module, which is not secure. Use the following method only with trusted models:

```
torch.save(model, "model.pth")
```

Another option is to use **`torch.onnx.export`** to export the model to an ONNX graph file. For more information, see [Introduction to ONNX](#). This file can then be loaded and run with the ONNX runtime, or with IBM Deep Learning Compiler (zDLC). The latter is based on ONNX and is described in another section of this document. For more information, see [Introduction to ONNX](#).

3.3 Model loading and inference

After a model is created and saved in a file, another program can load it from disk and run it. This section describes model loading and inference.

3.3.1 Model loading

Loading a saved `state_dict` from a file requires first instantiating an empty version of the model class, then loading the saved weights into the model (see Example 3-3).

Example 3-3 Instantiating and loading

```
import torch
# Expecting NeuralNetwork has been declared or imported from its previous
definition
model = NeuralNetwork() model.load_state_dict(torch.load('model.pth',
weights_only=True)) model.eval()
```

If the full model was saved, not just the `state_dict`, they can both be load both with a single call. However, use this method with *only* trusted models because this method uses the Python pickle module, which can be unsafe if the model comes from an untrusted source. .

Example 3-4 Load a full PyTorch model from a file and prepare it for inference

```
import torch
model = torch.load("model.pth", weights_only=False) model.eval()
```

On IBM z16 servers and later, to use the AI acceleration capabilities of the IBM Telum series of processors, inform the PyTorch runtime to use the device by running the following command:

```
model.to('nnpa')
```

The `nnpa` device is defined to PyTorch by additional code that is provided by IBM in the IBM Z Accelerated for PyTorch container image. This device is not available in other versions of

PyTorch that are built for different processor architectures or outside the AI Toolkit for IBM Z and LinuxONE offerings.

Parameters (tensors) might be created for a particular device, such as a graphics processing card, which is commonly a CUDA-compatible device (see [CUDA semantics](#)), on a platform other than IBM Z. Loading the model on IBM Z might result in an exception if that device is not available. The `torch.load` function supports a parameter that you can use to change the device so that you can use the model on IBM Z (for example, `map_location='cpu'`).

Before running your first inference, invoke `model.eval()` to set the internal state in the model for inference mode. Otherwise, you might get inconsistent results.

3.3.2 Inference

To perform inference on the model loaded, start it as a function (see Example 3-5). Also use the `torch.no_grad` context manager so that PyTorch does not need to compute gradients. Because there is a trained model, those computations can slow down the inference calculations.

Example 3-5 Invoked as a function

```
with torch.no_grad():  
    pred = model(x)
```

Alternatives to this approach are described in the PyTorch Documentation (see [PyTorch inference_mode](#)).

3.3.3 Downloading open-Source models

PyTorch models that are trained on other platforms typically run on IBM Z so that you can use popular open-source models such as Hugging Face (<https://huggingface.co>). Issues related to IBM Z being a big-endian architecture can occur and must be resolved through support channels. If feasible, retrain the model on IBM Z as a potential workaround.

There are several model repositories on the Internet, and several machine learning packages provide direct access to loading models. Download a model built specifically for PyTorch, or use an intermediate format such as ONNX to convert it into a PyTorch-compatible model. Hugging Face is a popular option that offers a wide range of PyTorch models. Hugging Face also hosts the Transformers library, which is widely used for Natural Language Processing (NLP) applications. For more information, see [Hugging Face Transformers](#).

As with any resource obtained from the Internet, ensure that your usage complies with applicable license terms and follows your enterprise's legal, security, and other organizational policies.

3.3.4 Model profiling

Profiling your PyTorch application reveals metrics about the performance and architecture of your model. This data is captured in JSON format and can be processed using PyTorch APIs or graphical tools. Both can help you determine if there are bottlenecks in your model and to diagnose performance problems.

Model profiling is covered in more detail in the following PyTorch tutorials:

- ▶ [PyTorch Profiler](#)
- ▶ [Visualizing Models, Data, and Training with TensorBoard](#)

You can capture profiling information by using the **torch.profiler** module before and after the targeted section of code to profile (usually with training or inference), as shown in Example 3-6.

Example 3-6 Capturing profiling information

```
prof = torch.profiler.profile(...)
prof.on_trace_ready = torch.profiler.tensorboard_trace_handler("./trace")
prof.start()

# Code of interest

prof.stop()
```

Review the PyTorch tutorial for details on the arguments to pass to **profile.profiler.stop**, which writes the collected data to a JSON file in the trace directory in the examples shown in this chapter.

This file can be opened in the Chrome trace viewer (<chrome://tracing> in the Google Chrome browser) or in Tensorboard. The PyTorch trace example that is displayed in the Chrome trace viewer is shown in Figure 3-1.



Figure 3-1 PyTorch trace displayed in Chrome trace viewer

You can open the JSON log file in TensorBoard, but only limited CPU information is shown. Profiling with Tensorboard has more features, but it requires additional code beyond this simple example to be inserted into the application. Review the PyTorch tutorial for details.

3.4 Support for IBM Telum processors

To maximize PyTorch performance on IBM z16 servers and later, take advantage of the integrated Neural Network Assist (NNPA) instructions of the IBM Telum processor. The AI Toolkit for IBM Z and IBM LinuxONE container images include IBM enhancements that enable open-source libraries to use these hardware capabilities. PyTorch applications that run with the IBM Z Accelerated for PyTorch container image can use this hardware acceleration in a manner similar to how PyTorch applications use other AI accelerators on other platforms.

Existing PyTorch programs written in Python require that models or tensors be loaded onto the nnpa device by using techniques described in previous sections. To achieve the best results from a model on IBM Z, make minor modifications to the PyTorch application. These updates activate the IBM extension that enables PyTorch to use the Neural Network Assist instructions through the nnpa device as recognized by PyTorch.

3.5 Near real-time credit card fraud detection use case

In this use case, the task is to predict whether a credit card transaction is fraudulent based on information from the transaction and the previous six transactions. Because the input data is temporal, a Recurrent Neural Network (RNN) is used. To view the code for this use case, see [ibmz-accelerated-for-pytorch](#).

3.5.1 Background information

As discussed in Chapter 2, “Overview of fraud detection use case” on page 11, the dataset for the use case comes in the form of a comma-separated values (CSV) file that contains credit card transaction information. These transactions are sorted by user and date-time so that consecutive user transactions can be accessed. These sorted transactions are then split into training, validation, and testing sets.

The dataset contains far more nonfraudulent transactions than fraudulent ones. Therefore, the training set uses weighted sampling while producing batches to ensure that the model is trained with an equal distribution of both.

3.5.2 Model construction and training

PyTorch supplies various neural network constructs, including Gated Recurrent Unit (GRU) and Long Short-Term Mean (LSTM) layers for RNN models, which are interchangeable for this use case.

Example 3-7 uses two RNN layers where the output of the first layer is directly fed into the second. The output of the second layer is then fed into a Dense layer for classification.

Example 3-7 Model construction and training

```
class RNNModel(torch.nn.Module):
    def __init__(self, rnn_type: str = 'lstm'):
        super().__init__()
        if rnn_type == 'lstm':
            rnn_module = torch.nn.LSTM
        else:
            rnn_module = torch.nn.GRU
        self.rnn = rnn_module(220, 200, num_layers=2, batch_first=True)
        self.fc = torch.nn.Linear(200, 1)
        self.sig = torch.nn.Sigmoid()
```

In Example 3-7, the way the forward method is coded ties together the operations in a logical sequence. The model is then trained for 20 epochs, using 50,000 equally distributed batches of examples from the training set per epoch. This can be done by using `credit_card_fraud_training.py`. To view the code, access the following URL:

https://github.com/IBM/ibmz-accelerated-for-pytorch/blob/main/samples/credit-card-fraud/credit_card_fraud_training.py

3.5.3 Model profiling

For profiling this model on the nnpa device, Figure 3-2 illustrates a difference in the operation that is taking advantage of an optimized execution path, which reduced the number of operations required to support the LSTM operation.

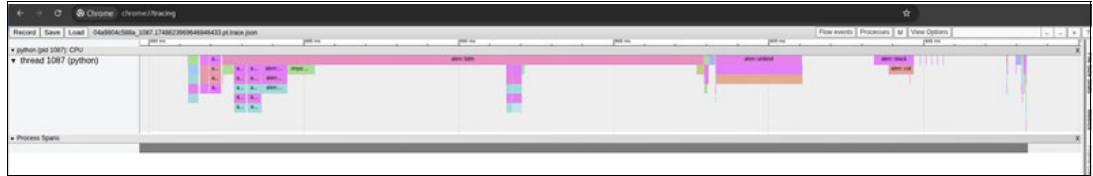


Figure 3-2 Profile of the model run on the nnpa device



IBM Z Deep Learning Compiler

As AI adoption accelerates across enterprise workloads, moving AI solutions from prototype to production presents a variety of technical challenges. These challenges include ensuring secure and efficient data access, meeting performance and service level agreements (SLAs), enforcing model governance policies, and implementing robust monitoring and observability capabilities.

Many of these challenges stem from the mismatch between the environments used for AI model development and training and those used for deployment. AI solution development often occurs in flexible, unconstrained environments, whereas production systems must adhere to strict operational, compliance, and infrastructure constraints.

The IBM Z AI strategy addresses these challenges by enabling model development and training on any platform, whether on IBM Z or in external environments such as cloud-based AI services. This flexibility allows teams to use existing toolchains and infrastructure investments. After models are trained, IBM Z provides a streamlined path for deployment, offering seamless portability onto IBM Z. This approach ensures that AI workloads integrate into mission-critical systems with the performance, reliability, and security that enterprise environments demand.

One of the key technologies that IBM uses to meet this requirement on the IBM Z platform is the Open Neural Network Exchange (ONNX). To enable ONNX model usage on the IBM Z platform, IBM Z provides the IBM Z Deep Learning Compiler (zDLC). The compiler allows customers to convert their ONNX models into shareable binary files that can then be integrated within their AI workloads. For more information, see [ONNX](#) and [IBM Z Deep Learning Center](#).

This chapter guides you through the IBM Z Deep Learning Compiler and its role in supporting the IBM Z strategy for AI adoption.

This chapter includes the following topics:

- ▶ 4.1, “Introduction to ONNX” on page 26
- ▶ 4.2, “Introduction to IBM Z Deep Learning Compiler” on page 30
- ▶ 4.3, “Compiling a model with IBM Z Deep Learning Compiler” on page 32
- ▶ 4.4, “Model loading and inferencing using compiled objects” on page 34
- ▶ 4.5, “AI enablement on IBM Z using zDLC” on page 35

- ▶ 4.6, “Overview of zDLC deployment scenarios and exploiters” on page 37
- ▶ 4.7, “Use Case: Near Real-Time Credit Card Fraud Detection” on page 41
- ▶ 4.8, “IBM Z Deep Learning Compiler Features” on page 42
- ▶ 4.9, “IBM Z Deep Learning Compiler Telum Support” on page 42
- ▶ 4.10, “IBM Z Deep Learning Compiler LLM enablement” on page 44
- ▶ 4.11, “IBM Z Deep Learning Compiler quantization support” on page 45
- ▶ 4.12, “IBM Z Deep Learning Compiler debug instrumentation” on page 48
- ▶ 4.13, “IBM zDLC performance features” on page 48
- ▶ 4.14, “IBM Z Deep Learning Compiler scope and versioning” on page 49

4.1 Introduction to ONNX

ONNX is a pivotal component of the IBM Z Deep Learning Compiler (zDLC) because it enables seamless interoperability and deployment of AI models on IBM Z systems. ONNX provides a standardized format for representing machine learning models. Models trained in frameworks such as PyTorch or TensorFlow can be converted and optimized by zDLC to run on IBM Z hardware, which can include the Telum processor with its AI accelerator. This portability streamlines the workflow, supporting IBM’s Build and Train Anywhere, Deploy on IBM Z strategy and aligning with a broad open-source ecosystem. By using ONNX, zDLC enhances performance for mission-critical workloads and facilitates efficient model serving in enterprise settings, such as through Watson Machine Learning for z/OS.

This section guides you through ONNX models, followed by a high-level overview of ONNX model converters and visualizers for deep learning models.

Deep learning with neural networks relies on computation graphs. A computation graph is a directed graph in which nodes represent operations or variables and edges depict data flow. Frameworks such as CNTK, Caffe2, Theano, and TensorFlow use static graphs, whereas PyTorch and Chainer employ dynamic graphs. Both types offer interfaces that simplify graph construction and provide optimized runtimes for processing. These runtimes enable efficient modeling of complex computations, such as those in machine learning, by breaking them into smaller steps.

The computation graph acts as an intermediate representation (IR) that captures the developer’s source code intent, enabling optimization and translation for running on devices such as CPUs, GPUs, or FPGAs.

[ONNX](#) was created as a common intermediate representation (IR) that fosters a robust open ecosystem. ONNX was created to make AI more accessible so that developers can choose the best framework for their project at any development or deployment stage,

ONNX is an open standard format that empowers AI developers to choose the right tools as their project evolves. ONNX provides an open-source format for AI models, both deep learning and traditional machine learning. It defines an extensible computation graph model, and provides definitions of built-in operators and standard data types.

ONNX is a widely supported format and is available in many frameworks, tools, and hardware platforms (see [ONNX Supported Tools](#)). Enabling interoperability between different frameworks and streamlining the path from research to production increases the speed of innovation in the AI community. There are two ONNX variants:

- ▶ The neural-network-only ONNX variant recognizes tensors as input and output types (see [Operators.md](#)).
- ▶ The classic machine-learning ONNX-ML variant also recognizes sequences and maps (see [Operators-ml.md](#)).

ONNX-ML extends the ONNX operator set with machine-learning algorithms that are not based on neural networks. In this publication, the focus is on the neural-network-only ONNX variant, referred to simply as ONNX.

The ONNX standard offers a unified format for transferring models between major machine-learning frameworks. It uses the Protocol Buffers definition language for its syntax (see [ONNX Documentation](#)), with the ModelProto as the top-level definition (see [ModelProto](#)). ONNX supports approximately 200 graph operators, enabling a wide range of machine-learning and deep-learning primitives, including convolution, LSTM, and matrix multiplication.

ONNX provides framework interoperability so that you can develop in your preferred framework without worrying about downstream inference implications (see [ONNX Build Model](#)). ONNX simplifies access to hardware optimizations. ONNX-compatible runtimes and libraries are designed to maximize performance across hardware platforms (see [ONNX Deploy Model](#)). ONNX is regularly updated with new versions (see [ONNX Releases](#)). Each version maintains compatibility with an earlier version to help ensure that models created with earlier versions remain usable.

4.1.1 Overview of ONNX model converters and visualizers

The ONNX community offers tools to streamline the creation and deployment of deep learning models. It also provides utilities for optimizing the ONNX models for size, accuracy, resource usage, and performance. Popular tools, such as Netron, are available for visualizing ONNX models. For more information, see [Supported Tools](#) and [optimizer](#).

Converting models to ONNX includes the following advantages:

- ▶ Cross-platform compatibility: ONNX provides a standard format for representing machine learning models, which can make it easier to deploy models across different frameworks such as PyTorch or TensorFlow. You can train models in one framework and deploy them in another framework that supports ONNX runtime.
- ▶ Improved performance: ONNX runtime optimizes models for inferencing by applying various hardware and software-specific optimizations, such as graph optimizations. Also, it supports execution on diverse hardware, such as CPUs and GPUs, ensuring efficient usage of resources.
- ▶ Interoperability: ONNX provides a way to train models, such as P

You can convert machine learning models from various frameworks. Supported frameworks include the following conversion tools:

- ▶ [sklearn-onnx](#): converts models from [scikit-learn](#)
- ▶ [tensorflow-onnx](#): converts models from [tensorflow](#)
- ▶ [onnxmltools](#): converts models from [lightgbm](#), [xgboost](#), [pyspark](#), [ibsvm](#)
- ▶ [torch.onnx](#): converts model from [pytorch](#)

ONNX converters include the following limitations:

Framework-specific converters: ONNX converters are not interoperable across frameworks. For example, `tensorflow-onnx` is exclusive to TensorFlow, `sklearn-onnx` is tailored for scikit-learn, and `torch.onnx` is specific to PyTorch. This limitation restricts cross-framework compatibility.

Customization challenges: Supporting custom components in machine-learning models is complex. Custom layers or functions often require dedicated converters, which essentially duplicate the prediction function implementation. Deep-learning frameworks such as PyTorch

or TensorFlow simplify this process when custom components use their native primitives, as these might often execute across different environments with minimal effort. However, scikit-learn presents additional challenges because it relies on NumPy or SciPy for operations such as addition or multiplication. You must reimplement scikit-learn transformers or predictors by using ONNX primitives, even if they were originally built with NumPy.

4.1.2 Visualizing an ONNX model

Visualizing an ONNX model enhances understanding, debugging, optimization, and collaboration, ensuring robust deployment on platforms like IBM Z with zDLC. Tools like Netron make this process accessible by providing clear, interactive representations of the model's computation graph.

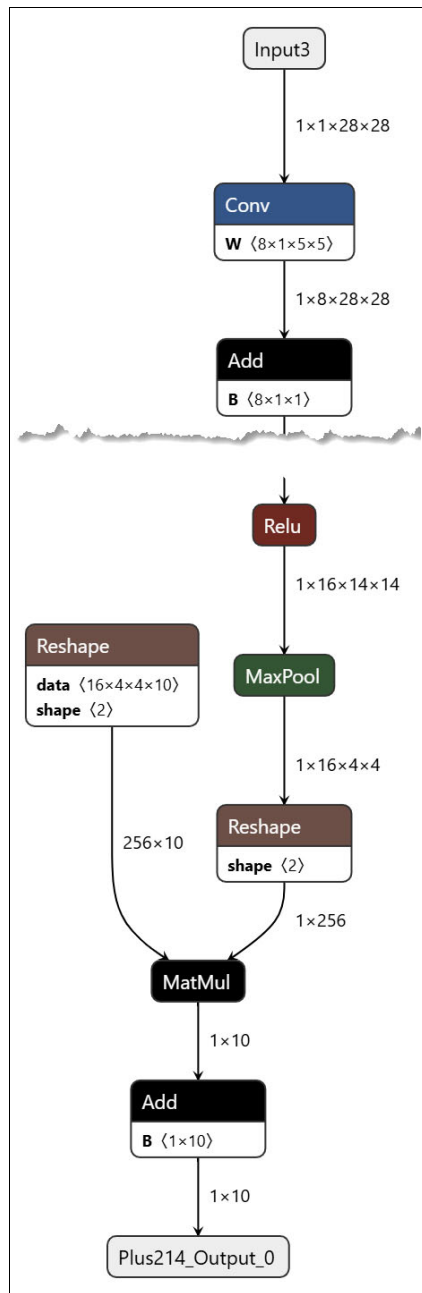


Figure 4-1 A portion of the sample ONNX visualization

Figure 4-1 on page 28 shows a portion of an AI model visualization from the [mnist-8.onnx](#) model, which is created by using the [Netron](#) application. Netron examines each node in the model, including inputs, outputs, weights, and other attributes. For complex models, this tool is highly valuable, providing a basis for comparing pre- and post-conversion models.

4.1.3 Pytorch to ONNX model export

The [torch.onnx](#) module provides APIs to capture the computation graph from a native PyTorch model and convert it into an [ONNX graph](#). The exported model can be consumed by any of the many [runtimes that support ONNX](#), including IBM's [IBM Z Deep Learning Compiler](#). For more information, see [Exporting a PyTorch model to ONNX natively using PyTorch](#).

The ONNX exporter API offers two variants for use. Figure 4-2 shows sample code to export a simple model.

```
import torch

class MyModel(torch.nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 128, 5)

    def forward(self, x):
        return torch.relu(self.conv1(x))

input_tensor = torch.rand((1, 1, 128, 128), dtype=torch.float32)

model = MyModel()

torch.onnx.export(
    model,                    # model to export
    (input_tensor,),          # inputs of the model,
    "my_model.onnx",          # filename of the ONNX model
    input_names=["input"],    # Rename inputs for the ONNX model
    dynamo=True              # True or False to select the exporter to use
)
```

Figure 4-2 Sample ONNX exporter code

4.1.4 TorchDynamo-based ONNX Exporter

The [TorchDynamo](#) engine hooks into Python's frame evaluation API to dynamically capture and translate bytecode into an [FX graph](#). This FX Graph is then optimized before being converted into an ONNX graph for further processing or deployment.

The primary advantage of this approach is that TorchDynamo uses bytecode analysis to capture an FX Graph, preserving the model's dynamic behavior, such as control flow, unlike traditional static tracing methods.

4.1.5 Tensorflow to ONNX model export

TensorFlow models must be converted by using the open-source [tensorflow-onnx](#) package supported by the ONNX community. The following commands describe a broad outline of how

to convert various TensorFlow models to ONNX. For more information, see [Converting a TensorFlow model to ONNX using TensorFlow-ONNX](#).

To convert a TensorFlow model to ONNX, use the following command (on one line):

```
python -m tf2onnx.convert --saved-model <path_to_saved_model> --output  
<path_to_output_onnx_file>
```

For converting a TensorFlow Lite model, use the `--tflite` flag instead of `--saved-model` with the following command (on one line):

```
python -m tf2onnx.convert --tflite <path_to_tflite_model> --output  
<path_to_output_onnx_file>
```

Use other options to control the conversion process, such as setting the target ONNX opset version with the `--opset` argument in the following command (on one line):

```
python -m tf2onnx.convert --saved-model <path_to_saved_model> --output  
<path_to_output_onnx_file> --opset 21
```

4.1.6 Recommended practices for converting ONNX models for IBM Z deployment

To prevent endian-related issues, IBM recommends that you convert models to ONNX format on the same platform that is used for training. For example, convert a model trained in an x86 environment to ONNX on an x86 system before transferring it to IBM Z or LinuxONE.

To determine which ONNX opset to specify for model conversion using the IBM Z Deep Learning Compiler, see the following references:

- ▶ [NNPA supported opset level](#)
- ▶ [CPU supported opset level](#)

On IBM z16, z17, or LinuxONE 5 machines (or later), use the highest opset level specified under NNPA operator support. This is typically the first statement in the NNPA supported opset level file.

4.2 Introduction to IBM Z Deep Learning Compiler

This section summarizes how ONNX models fit within the IBM Z strategy of promoting the adoption of open-source standards for AI model creation. The section includes the strategy of enabling customers to extend their existing AI investments from other frameworks to IBM Z through ONNX models. It also reviews framework support for exporting models from popular environments to ONNX by using converters.

After an AI model is converted to ONNX format, it is ready for testing and production deployment on IBM Z or LinuxONE. At this deployment stage, you must use technology that enables the creation of an inference solution fit for IBM Z and LinuxONE workloads. These critical workloads frequently have low-latency requirements and depend on the tight colocation and interaction of multiple products, such as IBM CICS®, IBM Db2, IMS, and others. AI inference solutions must coexist and thrive in this environment, delivering predictions within SLA windows, as they run alongside these critical business workloads.

To meet the requirements for ONNX models, IBM Z and LinuxONE use an ONNX model compiler called IBM Z Deep Learning Compiler (zDLC), which builds on top of the [ONNX-MLIR](#) open-source project. This compiler uses Multi-Level Intermediate

Representation (MLIR) to convert an ONNX model from a .onnx file into a highly optimized shared-object library. During compilation, the IBM Z Deep Learning Compiler (zDLC) optimizes the inference graph and generates a library tailored for the latest IBM Z and LinuxONE hardware enhancements. The compiler output is a lightweight, optimized program that enables efficient inference based on the input ONNX model.

Figure 4-3 on page 32 provides an architectural diagram for the high-level transformation of an ONNX operation through the onnx-mlir compiler.

ONNX-MLIR defines five primary dialects: ONNX, Krnl, Affine, Std, and LLVM. Among these, ONNX and Krnl are the most recent and are designed to support ONNX model representation and transformation within the MLIR framework.

ONNX-MLIR organizes these dialects into four abstraction levels:

1. First abstraction level: Provides a high-level representation of ONNX operations. It includes operations in the onnx and std dialects. The onnx dialect is automatically generated by an importer implemented as a Python script.
2. Second abstraction level: Includes the Krnl, Affine, and Std dialects. The Krnl dialect serves as an intermediate representation tailored for loop transformations and optimizations. It facilitates affine transformations such as tiling, skewing, and permutation, making it well suited for optimizing computational loops. Krnl acts as a bridge between the high-level ONNX dialect and the lower-level Affine, Std, and LLVM dialects, enabling efficient lowering and code generation.
3. Third abstraction level: Includes the Affine and Std dialects, where existing optimization passes in MLIR can be freely applied.
4. Fourth abstraction level: Consists solely of the LLVM dialect, which is responsible for generating LLVM IR and producing optimized bitcode for target architectures.

MLIR passes in ONNX-MLIR facilitate both dialect-to-dialect conversion and intra-dialect optimization. In the compilation pipeline, the ONNX dialect is first lowered to the Krnl dialect by using the `--convert-onnx-to-krnl` pass. Next, the Krnl dialect is partially transformed into the Affine and Std dialects through the `--convert-krnl-to-affine` pass, leaving some Krnl operations in place. These remaining Krnl operations, together with Affine and Std operations, are then lowered to the LLVM dialect by using the `--convert-all-to-llvm` pass. This final stage generates optimized LLVM IR and bitcode to support efficient execution on architectures such as IBM Z.

The right side of Figure 4-3 on page 32 shows optimization passes that can be applied at each abstraction level. This section lists only key optimization passes. The list is not exhaustive.

Section 4.3, “Compiling a model with IBM Z Deep Learning Compiler” on page 32 describes various deployment scenarios and describes how the IBM Z Deep Learning Compiler performs without locking you into any framework or operating system within the IBM Z platform.

The compiled models take advantage of IBM Z technologies, including SIMD on IBM z13 and later, and the Integrated Telum Accelerator for AI that is available on IBM z16 and z17, without requiring changes to the original model.

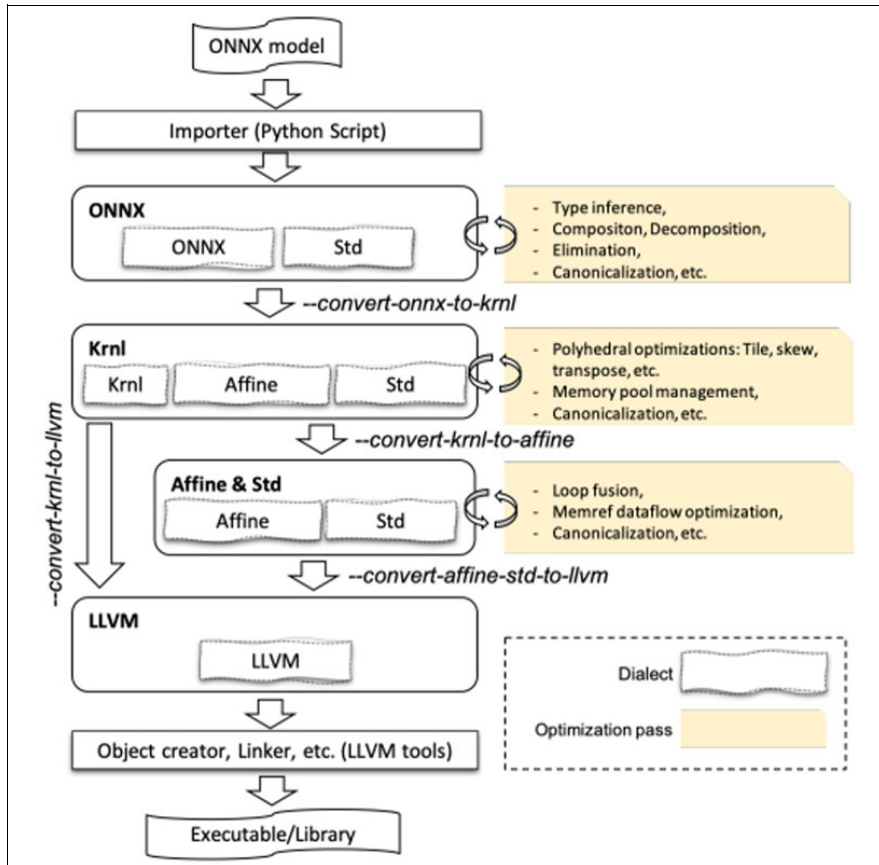


Figure 4-3 Compilation flow

4.3 Compiling a model with IBM Z Deep Learning Compiler

This section shows an example of image recognition that uses an ONNX MNIST model. In the example shown in Figure 4-4 on page 33, a user draws a number from 0 to 9 and receives output from the ONNX model. To view the digit recognition demonstration for the mnist ONNX mode, see [MNIST, handwritten digit prediction](#).

The ONNX MNIST model takes a 28×28 pixel image of a hand-drawn digit, processes it through a series of layers (MaxPool, Gemm, ReLU, and Softmax), and outputs a probability distribution that represents the digit. In this example, the model correctly predicts the digit “3” with the highest probability, indicated by the blue line next to the digits on the right side of the figure.

To run the same digit recognition ONNX model on IBM Z, the IBM Z Deep Learning compiler compiles the MNIST model to a shared library object depending on the following target application, CPU/AI Accelerator, IBM Z machine series, and the operating system based on the customers deployment scenarios (see [MNIST - Handwritten Digit Recognition](#)). For more information, see [IBM zDLC](#).

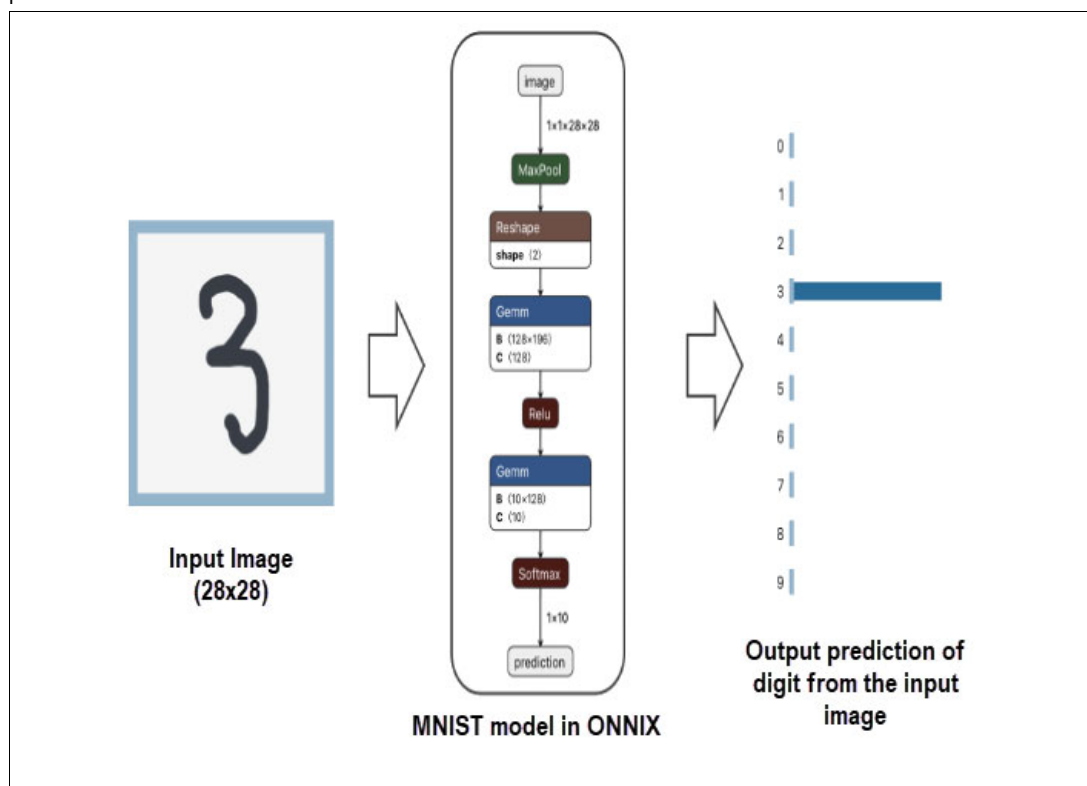


Figure 4-4 Using the ONNX MNIST model to predict the digit in the drawn image

To create, convert, or download an ONNX model, start by downloading the MNIST model, [mnist-12.onnx](#), as shown in Example 4-1. This example uses \$ZDLC_MODEL_DIR as the model directory, and \$ZDLC_MODEL_NAME specifies the model's name (without the .onnx) in that directory.

Example 4-1 Download the ONNX model file for the MNIST dataset

```
ZDLC_MODEL_NAME=mnist-12
wget --directory-prefix $ZDLC_MODEL_DIR
https://github.com/onnx/models/raw/main/validated/vision/classification/mnist/model1/$ZDLC_MODEL_NAME.onnx
```

Perform the following steps:

1. Download the zDLC image from [Container Images for IBM Z and LinuxONE](#).
2. Determine the version of the zDLC image to download.
3. Select the Getting started tab and use the listed steps to obtain an APIkey.
4. Log in to the icr.io registry by using the APIKey (see Example 4-2).

Example 4-2 Command to log in to the icr.io registry

```
docker login -u iamapikey icr.io
Password: <APIKey>
```

5. Set the ZDLC_IMAGE based on the IBM zDLC version by using the following command:
ZDLC_IMAGE=icr.io/ibmz/zdlc:5.0.0

6. Pull the image by using the following command:

```
docker pull ${ZDLC_IMAGE}
```

Use this image to compile a shared library of the model for your desired deployment use case.

Example 4-3 and Example 4-4 demonstrate the processes for generating shared libraries on Linux running on an IBM Z, with different configurations targeting the z16 CPU and the Network Processing Assist (NNPA) accelerator on the z17 Telum II processor.

Example 4-3 is the Docker command that is used to run a containerized process for compiling or optimizing a machine learning model (in ONNX format) on a Linux system running on IBM Z (specifically an IBM z17).

Example 4-3 Generate a shared library for an optimized ONNX model

```
docker run --rm -v ${ZDLC_MODEL_DIR}:/workdir:z ${ZDLC_IMAGE} --EmitLib --O3  
-march=z17 --mtriple=s390x-ibm-toz ${ZDLC_MODEL_NAME}.onnx
```

Example 4-4 is a Docker command for compiling or optimizing an ONNX machine learning model on a Linux system running on an IBM z17 with a focus on using the NNPA feature of the Telum processor.

Example 4-4 Generate a shared library for an optimized ONNX model with NNPA acceleration

```
docker run --rm -v ${ZDLC_MODEL_DIR}:/workdir:z ${ZDLC_IMAGE} --EmitLib --O3  
-march=z17 --mtriple=s390x-ibm-toz --maccel=NNPA ${ZDLC_MODEL_NAME}.onnx
```

Note: If you are running on an IBM z16, use `-march=z16` and `--maccel=NNPA` and access the IBM z16 Telum processor.

4.4 Model loading and inferencing using compiled objects

After the .onnx model is compiled into a shared library, a .so object file for the model is generated. You can then load this shared object in the inference application according to your business logic.

Figure 4-5 on page 35 shows a detailed screenshot of loading the compiled shared-library (.so) object and performing inference by using a Python inference program. For more information, see [IBM ZDLC](#).

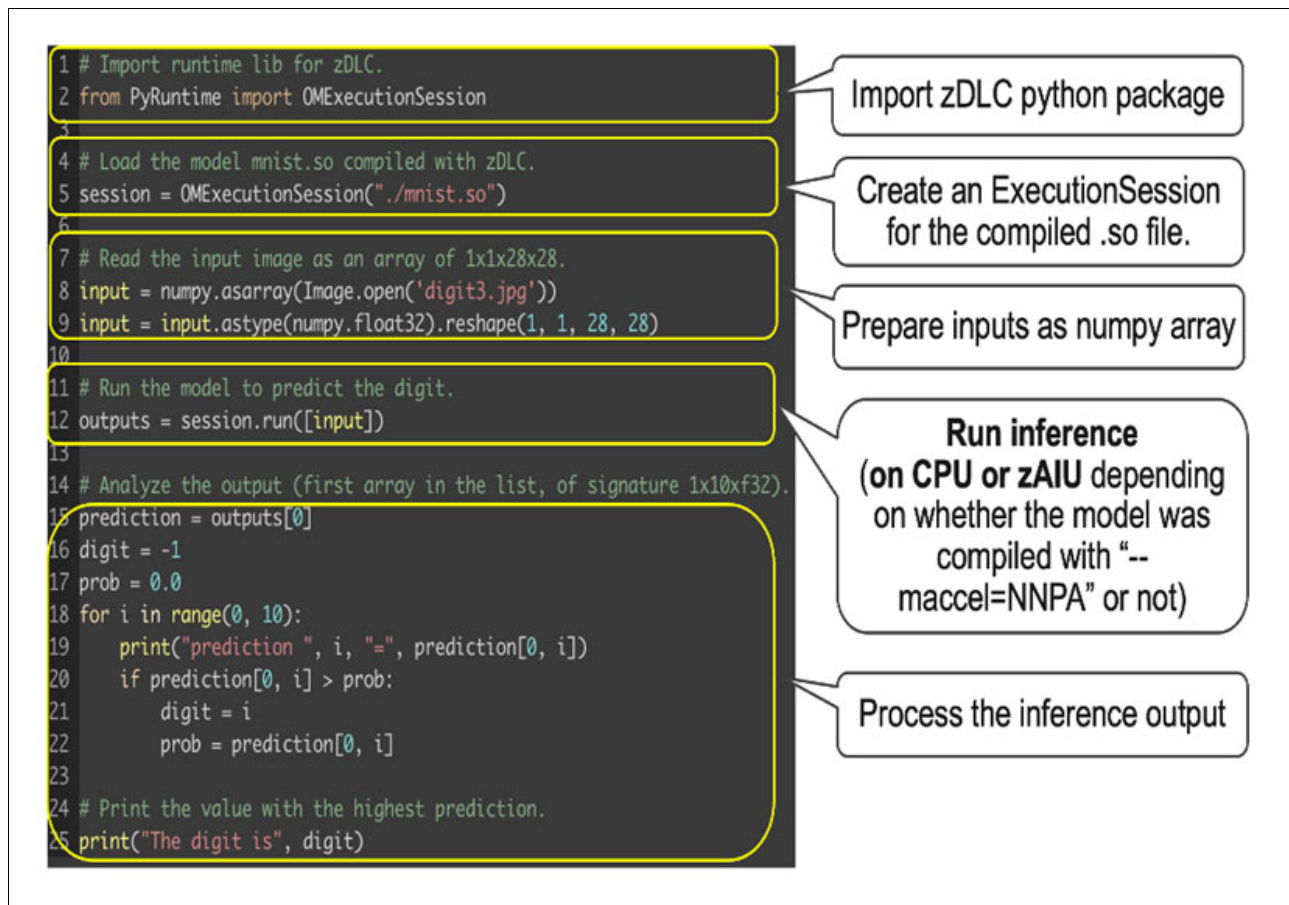


Figure 4-5 Python script for running inference on an ONNX model using IBM zDLC

4.5 AI enablement on IBM Z using zDLC

As businesses grow, the constant evolution of workloads presents recurring challenges. The pressure to integrate AI workloads into existing and new enterprise environments that deliver real-time insights while maintaining performance requirements is ever-present. AI workloads demand more from enterprise systems, and organizations are feeling the strain.

To meet these demands, a performant, scalable, low-latency, and highly available environment is essential, one that can turn investments in AI use cases into growth opportunities and provide real-time AI insights with trust and transparency for your business.

The IBM Z Deep Learning Compiler (zDLC) provides a path forward to integrate AI competency into your workloads while continuing to meet low-latency requirements for high-volume workloads and addressing security concerns related to data privacy for AI model usage in enterprise environments.

As a key strategic solution for AI acceleration on IBM Z, the IBM Z Deep Learning Compiler offers various benefits. It addresses customer challenges and provides many options that you can use to exploit the features available in the compiler.

The following list summarizes key features of the IBM Z Deep Learning Compiler (zDLC):

- Build and train AI models anywhere: The rapid growth of AI led to the emergence of numerous frameworks and cloud providers that offer integrated development environments, enabling users to develop and train machine-learning models, including deep-learning models.

By using the IBM Z Deep Learning Compiler (zDLC), you can use your existing investments in AI platforms and frameworks while integrating your AI use cases into enterprise workloads on IBM Z. With zDLC, you can export trained models from your preferred framework or platform in ONNX format, then compile those ONNX models for IBM Z systems. This flexibility enables you to develop and train AI models on any platform without being tied to a specific framework or ecosystem. (see Figure 4-6).

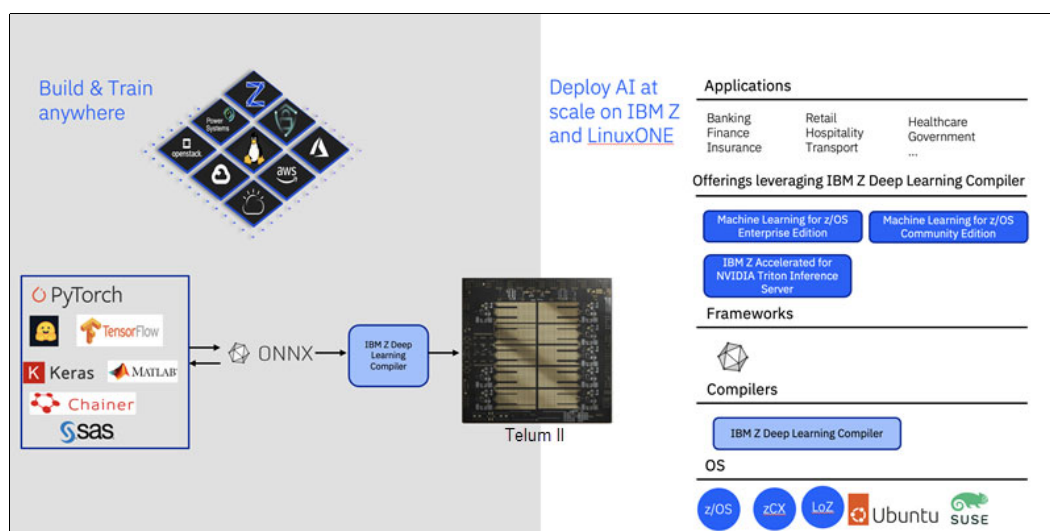


Figure 4-6 Workflow for building, training, and deploying AI models on IBM Z with IBM zDLC

- Deploy and integrate existing and new deep learning models into IBM Z enterprise workloads

The IBM Z Deep Learning Compiler (zDLC) enables you to easily integrate AI solutions into enterprise workloads that target deep-learning models. The executable file generated by zDLC contains an entry point that can be called by languages such as C, C++, Java, Scala, and Python running on Linux on IBM Z, zCX, and z/OS on IBM z13 and later machines. The zDLC models include the following features:

- zDLC-compiled models are portable between various Z Platforms,
- No package or framework dependencies on the compiled models when using the compiled models via zDLC for inferencing.
- Runtime interfaces to integrate compiled models into applications are available by default via the zDLC container.
- This flexibility of deployment and options for integration gives the enterprise customers portability option within the platforms on IBM Z itself.
- Ahead-of-Time (AOT) compilation: AOT compilation reduces startup latency compared to Just-in-Time (JIT)-based frameworks.

- Bringing AI to data

AI inferencing capabilities on IBM z17 are powered by a second-generation on-chip AI accelerator that is built into the IBM Telum II processor. The accelerator features increased

frequency, enhanced compute capacity, and a 40 percent larger cache, enabling more than 450 billion inferencing operations per day with a one-millisecond response time.

With integrated support for Telum II, AI primitives in the IBM Z Deep Learning Compiler (zDLC) are designed to effectively exploit the Telum II Integrated Accelerator for real-time AI inferencing. The compiler supports low-latency AI requirements for enterprise workloads and includes the following benefits:

- With IBM z17, you can process up to 5 million inference operations per second, which is approximately 50 percent more than on IBM z16 by using zDLC.
- With IBM z17, you can process up to 450 billion inference operations per day, which is approximately 50 percent more than on IBM z16 by using zDLC.

Note: For more information on the IBM z17, see [IBM z17: The First Mainframe Fully Engineered for the AI Age](#)

► IBM zDLC and AI integration

IBM zDLC enables seamless AI integration across IBM Z environments, offering flexibility and broad compatibility. It is integrated into various products that provide inferencing solutions and supports multiple operating systems beyond RHEL, SUSE, and Ubuntu, including Linux on IBM Z, LinuxONE, zCX, and z/OS.

Figure 4-7 shows deployment options for AI inferencing on IBM Z:

- Triton Inference Server on Linux on Z for customizable deep learning.
- WMLz Online Scoring Community Edition on zCX for rapid REST-based scoring.
- WMLz Enterprise Edition on z/OS for native integration with advanced features and CICS/COBOL applications.

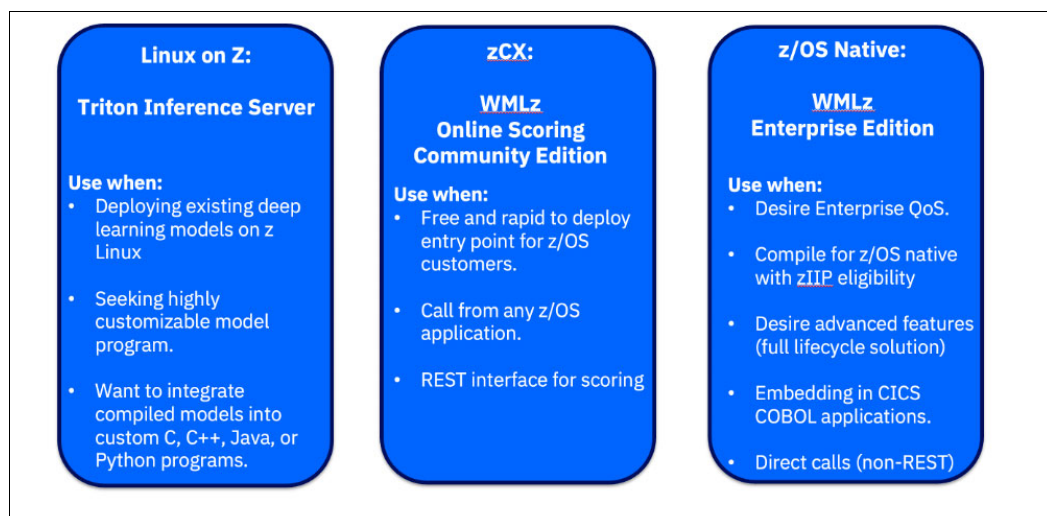


Figure 4-7 Deployment options for AI inferencing on IBM Z

4.6 Overview of zDLC deployment scenarios and exploiters

As discussed in section 4.3, “Compiling a model with IBM Z Deep Learning Compiler” on page 32, the IBM Z Deep Learning Compiler offers tailored options for compiling an ONNX model based on the target deployment scenario. This section examines the ONNX model

compilation workflow and deployment setup. It also describes the use of IBM Z Deep Learning Compiler to produce a compiled AI model for specific deployment scenarios.

Figure 4-8 describes the end-to-end AI workflow with the IBM Z Deep Learning Compiler.

With IBM zDLC, you can build and train your AI models on either IBM Z, x86, or PowerPC Little Endian (PPCle) platforms. The platforms can run either on premises or in a cloud, and you can convert or export your models to an onnx model.

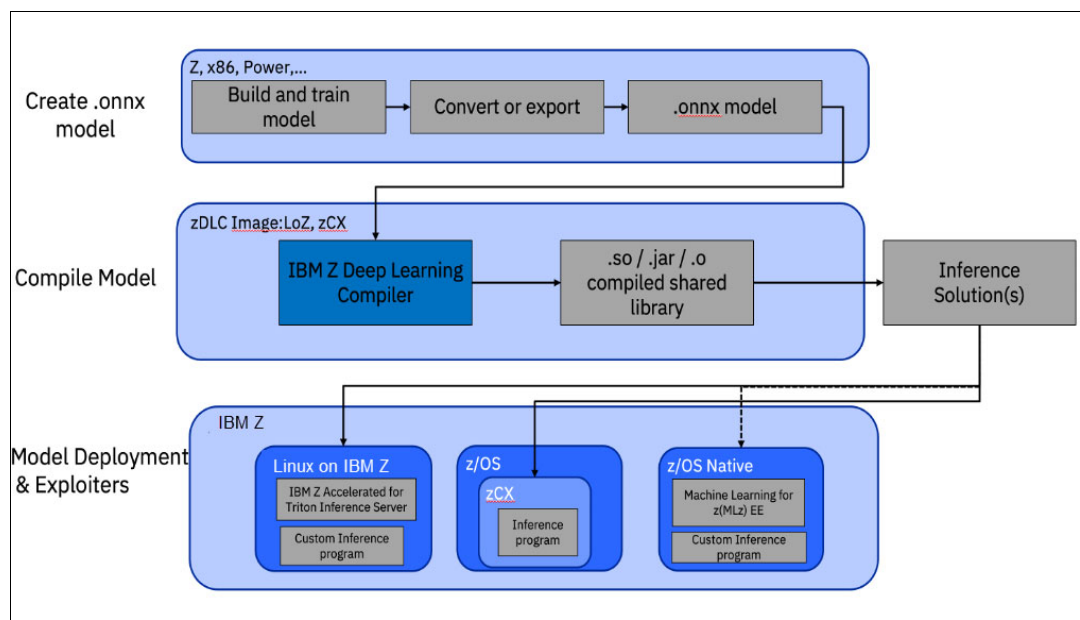


Figure 4-8 End-to-end AI workflow on IBM Z

A zDLC deployment includes several other key features. Compiling the ONNX model into either an .so or .jar or .o shared library requires the IBM Z Deep Learning Compiler. The decision of which shared library to generate depends on many variables:

- ▶ IBM Z operating systems
- ▶ The target IBM Z CPU vs Telum Accelerator
- ▶ Application language

Table 4-1 on page 39 shows three sample IBM Z operating systems, including Linux on IBM Z, z/OS Container Extensions (zCX), and z/OS. Each has its own deployment scenario and compilation options. The underlying IBM Z hardware on which they run can be z14, z15, z16, or z17.

The generated binary can be targeted to run on only the CPU or to use the Telum and Telum II integrated AI accelerators that are available on IBM z16 and z17.

Table 4-1 on page 39 describes how these deployments are supported by the IBM Z Deep Learning Compiler. Use the following key for Table 4-1 on page 39:

```
ZDLC_IMAGE=icr.io/ibmz/zdlc:5.0.0
ZDLC_MODEL_DIR= <The local directory where all the ONNX models are stored>
ZDLC_MODEL_NAME= mnist-12
NNPA {Neural Network Processing Assist} Telum I/II accelerator [Review Once]
```

Table 4-1 Sample compile options for different operating system and deployment scenarios

Operating System	Deployment Scenario	Compilation Options
Linux on IBM Z CPU	Inference via C++ runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z16 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitJNI --03 -march=z16 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z16 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via C++ runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z17 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitJNI --03 -march=z17 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z17 --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
IBM z/OS CPU	Inference via C++ runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z16 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z16 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z16 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via C++ runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z17 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z17 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z17 --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
Linux on IBM Z NNPA	Inference via C++ runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z16 --maccel=NNPA --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitJNI --03 -march=z16 --maccel=NNPA --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z16 --maccel=NNPA --mtriple=s390x-ibm-loz \${ZDLC_MODEL_NAME}.onnx</code>

	Inference via C++ runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z17 -maccel=NNPA --mtriple=s390x-ibm-1oz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Java runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitJNI --03 -march=z17 -maccel=NNPA --mtriple=s390x-ibm-1oz \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via Python runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitLib --03 -march=z17 -maccel=NNPA --mtriple=s390x-ibm-1oz \${ZDLC_MODEL_NAME}.onnx</code>
IBM z/OS NNPA	Inference via C++/Java/Python runtime, Deployed on z16	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z16 -maccel=NNPA --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>
	Inference via C++/Java/Python runtime, Deployed on z17	<code>docker run -v \${ZDLC_MODEL_DIR}:/workdir:z \${ZDLC_IMAGE} --EmitObj --03 -march=z17 -maccel=NNPA --mtriple=s390x-ibm-zos \${ZDLC_MODEL_NAME}.onnx</code>

After the models are compiled according to the deployment parameters discussed in Table 4-1 on page 39, they can be integrated into a home-grown inferencing solution or used with IBM Z products that incorporate IBM Z Deep Learning Compiler (zDLC) compiled models to provide ONNX model support.

The following sections describe two solutions that use the IBM Z Deep Learning Compiler

4.6.1 IBM Z Deep Learning exploiter on Linux for IBM Z

The IBM Z Accelerated Triton Inference Server provides an `onnxmlir` backend, which allows customers to use Triton Inference Server for serving ONNX models. The `onnxmlir` backend supports the deployment of zDLC-compiled ONNX models (`model.so`) with the Triton Inference Server.

For more information about the `onnxmlir` backend, see [onnxmlir-triton-backend](#). The Triton `onnxmlir` backend requires the compiled model to be named `model.so`.

The IBM Z Accelerated Triton Inference Server is an inferencing solution that targets users running on LinuxONE, Linux for IBM Z, and zCX platforms.

4.6.2 IBM Z Deep Learning exploiter on z/OS

The Machine Learning for IBM z/OS Enterprise Edition provides support for building, training, and deployment of ONNX models on the IBM z/OS operating system via an ONNX compiler service. The ONNX compiler service bundles the IBM zDLC as a service by bundling it as an IBM Machine Learning for z/OS (MLz) component. If you use IBM z/OS and use ONNX models with CICS® and COBOL programs, perform the following the steps:

1. [Configure](#) the ONNX compiler service for your MLz instance for IBM z/OS ([Configuring the ONNX compiler service for your ML for IBM z/OS](#)).
2. Import the ONNX model which is transparently compiled into a shared `.jar` object using the ONNX compiler service which can then run by using a Scala application (see [Importing an ONNX model](#)).

For more information, see [Overview of MLz Enterprise](#), [Configuring the ONNX compiler service for your ML for IBM z/OS](#), and [Importing an ONNX model](#).

After importing and saving the .onnx model, you can deploy the model, test the deployment, and run the scoring service for predictions. When using the IBM MLz Enterprise Edition, you can perform the following actions:

- ▶ Enable on-chip AI accelerator for your ONNX models on z16 and z17.
- ▶ Enable micro-batching inference for your ONNX models

With MLz, you can perform the following actions:

- ▶ Prepare an ONNX model for online scoring. For more information, see [Preparing a model for online scoring with CICS program ALNSCORE](#).
- ▶ Prepare an ONNX model for online scoring. For more information, see [Preparing a model for online scoring in COBOL program using a Deployment ID with WOLA](#).

4.7 Use Case: Near Real-Time Credit Card Fraud Detection

In this use case, the objective is to predict whether a credit-card transaction is fraudulent based on the information for that transaction and its previous six transactions. Because the input data is temporal, a Recurrent Neural Network (RNN) is used. The code for this use case follows the sample found on the following website:

https://github.com/IBM/zDLC/tree/main/code/credit_card_fraud_example

4.7.1 Background Information

As discussed in Chapter 2, “Overview of fraud detection use case” on page 11, the dataset for this use case is provided as a comma-separated values (CSV) file that contains credit card transaction information. These transactions are sorted by user and date/time, enabling consecutive user transactions to be accessed. The sorted transactions are then divided into training, validation, and testing sets.

Because the dataset contains many more nonfraudulent transactions than fraudulent ones, the training set uses weighted sampling when producing batches to ensure that the model trains with an equal distribution of both classes.

4.7.2 Training and Exporting a CCFD ONNX Model using PyTorch

To create the ONNX model for Credit Card Fraud Detection (CCFD) by using IBM zDLC, download the [Credit Card Fraud data set](#) and train the PyTorch model by using the [credit_card_fraud_training.py](#) script.

This process uses the Credit Card Fraud dataset to export and save the ccfid.onnx model in the ZDLC_MODEL_DIR directory from the trained PyTorch model. For more information, see [Running the Credit Card Fraud Detection Sample Program](#).

4.7.3 Building the CCFD .so using the IBM Z Deep Learning Compiler

Use the `--EmitLib` option to build a .so shared library of the CCFD model specified by ZDLC_MODEL_NAME in [Environment variables](#) as shown in Example 4-5.

Example 4-5 Command to build a .so shared library

```
docker run --rm -v ${ZDLC_MODEL_DIR}:/workdir:z ${ZDLC_IMAGE} --EmitLib --03
-march=z17 --mtriple=s390x-ibm-toz --maccel=NNPA ${ZDLC_MODEL_NAME}.onnx
```

4.7.4 Running the CCFD inference Python example

The sample program [credit_card_fraud_inference.py](#) calls the zDLC Runtime APIs by using [pybind](#) and [PyExecutionSession](#).

Copy the PyRuntime library out of the docker container by using the commands shown in Example 4-6.

Example 4-6 Copy the PyRuntime library

```
mkdir -p ${ZDLC_LIB_DIR}
docker run --rm -v ${ZDLC_LIB_DIR}:/files:z --entrypoint '/usr/bin/bash'
${ZDLC_IMAGE} -c "cp /usr/local/lib/PyRuntime* /files"
```

For details about two configuration approaches, see [Using Python interfaces](#). To avoid the creation of symbolic links, the example in this publication uses the PYTHONPATH approach.

Build the example Python image by using the command shown in Example 4-7.

Example 4-7 Build the image

```
docker build -f ${ZDLC_DIR}/docker/Dockerfile.python -t
zdlc-python-ccfd-example:latest .
```

Run the Python client with the command shown in Example 4-8.

Example 4-8 Run the clients

```
docker run --rm -v ${ZDLC_LIB_DIR}:/build/lib:z -v ${ZDLC_CODE_DIR}:/code:z -v
${ZDLC_MODEL_DIR}:/models:z --env PYTHONPATH=/build/lib
zdlc-python-ccfd-example:latest code/credit_card_fraud_inference.py
/models/${ZDLC_MODEL_NAME}.so
```

4.8 IBM Z Deep Learning Compiler Features

This section describes several IBM Z Deep Learning Compiler (zDLC) features so that you can take advantage of IBM Z hardware capabilities.

It first describes IBM Z Telum support enabled through IBM zDLC, followed by Large Language Model (LLM) support, including quantization features available on IBM z17 through the IBM zDLC 5.0.0 release.

This section also highlights the performance and debugging support available in the IBM Z Deep Learning Compiler.

4.9 IBM Z Deep Learning Compiler Telum Support

IBM z16 and z17 systems include an integrated accelerator for AI to enable real-time AI for transaction processing at scale. IBM z16 includes the Telum accelerator. IBM z17 has the Telum II accelerator. The IBM Z Deep Learning Compiler helps both new and existing deep learning models take advantage of this integrated accelerator for AI inferencing.

Any IBM Z can be used to compile models to take advantage of the Integrated Accelerator for AI, including IBM z15 and earlier machines. However, if acceleration is enabled at compile time, the compiled model runs on only an IBM Z with an accelerator. Machines with an accelerator can run models compiled without acceleration, but those models do not take advantage of the accelerator and are run on the CPU.

Similar to other compilers, the default settings in IBM zDLC compile models to run on the widest range of systems. To use machine-specific features, such as the IBM Integrated Accelerator for AI, specify an additional option when compiling the model.

When this option is set, supported ONNX operators are directed to the Telum accelerator instead of the CPU. The compile process handles routing operations between the CPU and accelerator and any required data conversion. No changes are required to your model.

Note: Telum and Telum II are processor names. The IBM Z Integrated Accelerator for AI is a component within those processors and is a hardware accelerator.

Note: Telum and Telum II are processor names. The IBM Z Integrated Accelerator for AI is a hardware component within those processors.

To compile a model that uses the Integrated Accelerator for AI, specify the `--maccel=NNPA` option when compiling the model to a shared library. Because the accelerator is available only on IBM z16 and later systems, use the `--march=z16` or `--march=z17` option when compiling.

Note: The options `--mcpu=z16` and `--mcpu=z17` are deprecated. Use the `--march` option for model compilation.

The command line to compile models that take advantage of the Integrated Accelerator for AI is shown in Example 4-9. For more information, see [Building a model .so using the IBM Z Deep Learning Compile](#).

Example 4-9 Compile the model

```
docker run --rm -v ${ZDLC_MODEL_DIR}:/workdir:z ${ZDLC_IMAGE} --EmitLib --O3
--march=z17 --mtriple=s390x-ibm-toz --maccel=NNPA ${ZDLC_MODEL_NAME}.onnx
```

After the model is compiled to use the IBM Z Integrated Accelerator for AI, you do not need to change any command-line options to run the model.

Example 4-10 copies the compiled model from the directory where the .so file is located to the target directory. It then prepares the model file by placing it in the code directory so that Docker can run the model.

Example 4-10 Copy then run the model.

```
cp ${ZDLC_MODEL_DIR}/${ZDLC_MODEL_NAME}.so ${ZDLC_CODE_DIR}
docker run --rm -v ${ZDLC_CODE_DIR}:/code:z ${GCC_IMAGE_ID}
/code/deep_learning_compiler_run_model_examples
```

The same compilation flags apply to shared libraries for any language, including Java and Python. Likewise, no additional steps are required when running these shared libraries.

All Telum operators use 16-bit numerical values (DFLOAT16). Different Telum operators require specific data layouts (for example, 3DS, 4DS) for optimal performance.

Offloading operators to the Telum accelerator involves the following steps:

1. Convert input data from `float32` to `dlfloat16`, a process informally called stickifying the data.
2. On Telum: Run the Telum-optimized operator with the converted `dlfloat16` data.
3. On CPU: Convert the output data from `dlfloat16` back to `float32`, a process informally called unstickifying.

The IBM zDLC selects operators that are suitable for running on Telum processors based on a cost model. zDLC optimizes stickify and unstickify operations by performing graph analysis to minimize CPU usage.

Note: Do not unstickify data if the consuming operator is running on Telum.

Figure 4-9 illustrates the workflow for optimizing an ONNX model using the IBM zDLC.

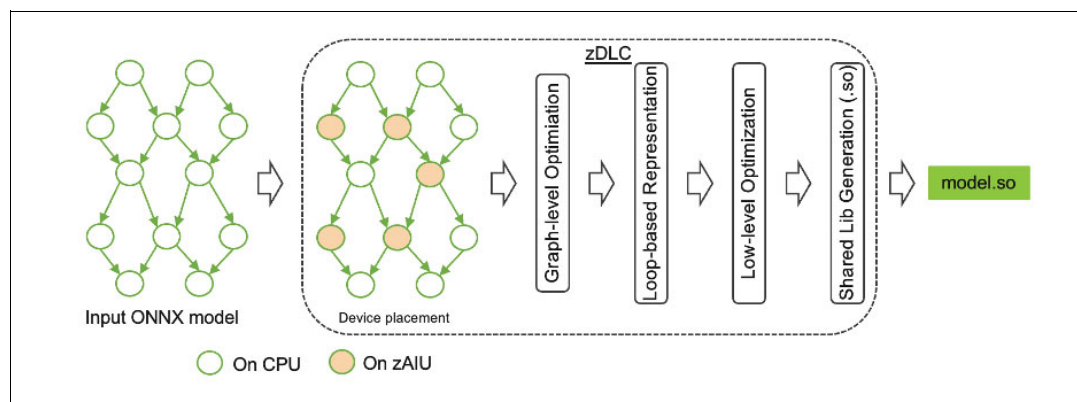


Figure 4-9 Workflow to optimize an ONNX model

4.10 IBM Z Deep Learning Compiler LLM enablement

The IBM zDLC supports running large language models (LLMs) such as IBM Granite®, RoBERTa, and GPT-2 on IBM Z systems.

A major requirement for running these LLMs is the need for substantial memory during model compilation.

When you use the compiler option `--store-constants-to-file`, constants are stored in a single binary file that has the same base name as the `model.so` file but has the suffix `.constants.bin` appended. This option is enabled by default when set to `true`.

This process generates the following files for each model:

- ▶ `model.so`: the compiled model file.
- ▶ `model.so.constants.bin`: the binary file containing constants.

Additional compiler options can control constant storage thresholds:

- ▶ `--constants-to-file-single-threshold` (value in KB, default = 1) controls the maximum size for a single constant.
- ▶ `--constants-to-file-total-threshold` (value in GB, default = 2) controls the total size of all constants to be stored.

For example, `--constants-to-file-single-threshold=1` and `--constants-to-file-total-threshold=1.5` instruct the compiler to store constants to a file if the total size of all constants is greater than 1 KB and the combined constant size exceeds 1.5 GB.

The `.constants.bin` file is mapped into memory the first time the inference is invoked. Example 4-11 demonstrates sample command use.

Example 4-11 Sample usage

```
docker run --rm -it -v ${ZDLC_MODEL_DIR}:/workdir:z ${ZDLC_IMAGE} --O3 --EmitLib
--mtriple=s390x-ibm-loz -march=z16 --maccel=NNPA ${ZDLC_MODEL_NAME}.onnx
--store-constants-to-file=false --constants-to-file-total-threshold=0.3
--onnx-op-stats TXT
```

4.11 IBM Z Deep Learning Compiler quantization support

Large language models (LLMs) are named for the number of parameters they contain. These models often include billions, and in some cases trillions, of parameters, primarily in the form of learned weights, which impose significant storage and memory-bandwidth requirements.

During inference, the model generates activations by multiplying input data by the weights. These activations can also be large, especially in very large models. Figure 4-10 shows this process visually: inputs flow through weighted connections to produce activations.

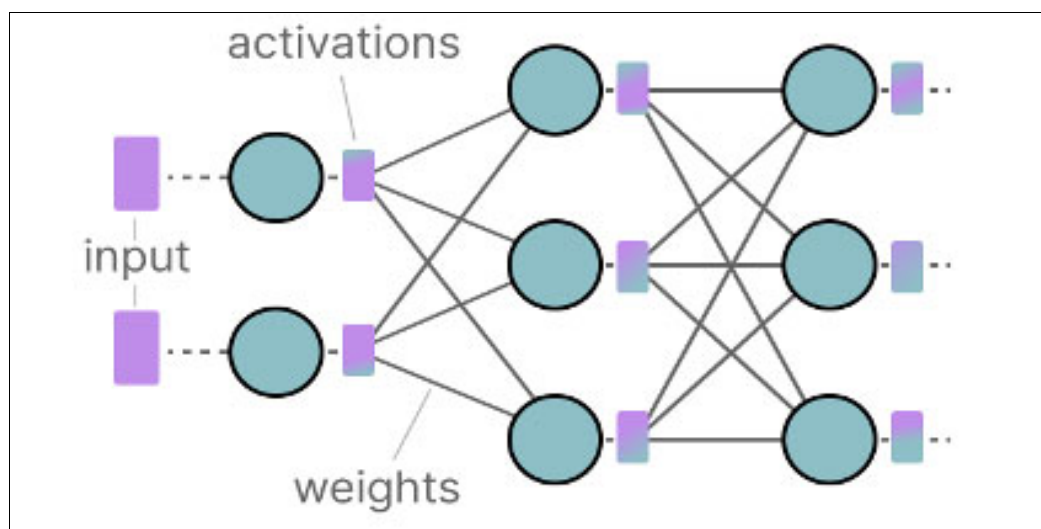


Figure 4-10 Flow from input to activations

Because these models contain billions of values, they must be stored as efficiently as possible to conserve space and reduce memory usage.

Quantization reduces the precision of a model's parameters from higher bit widths (for example, 32-bit floating point) to lower bit widths (for example, 8-bit integers). Some loss of precision or granularity often occurs when reducing the number of bits that are used to represent the original parameters. Figure 4-11 on page 46 visually explains the concept of quantization granularity in neural networks.

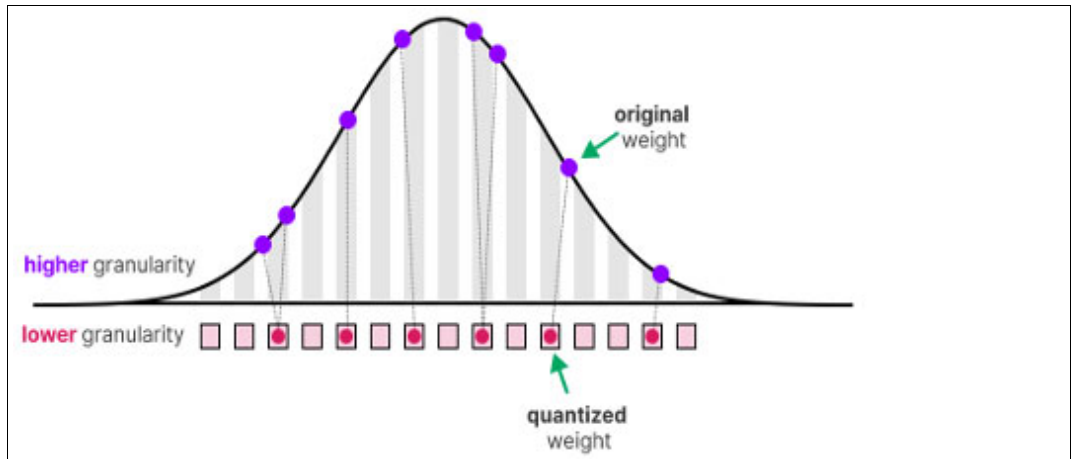


Figure 4-11 Quantization granularity

To illustrate this effect, take any image and represent it by using only eight colors.

A quantized image shows reduced color detail, and a zoomed-in area of the quantized image appears more grainy than the original because fewer colors are available to represent it. The main goal of quantization is to reduce the number of bits (or colors) needed to represent the original parameters while preserving their precision as much as possible.

For more information about this example, see [A Visual Guide to Quantization](#).

To illustrate this effect, we can take any image and use only 8 colors to represent it.

A quantized image shows reduced color detail and a zoomed-in area of the quantized image appears more “grainy” than the original because fewer colors are available to represent the image. The main goal of quantization is to reduce the number of bits (colors) needed to represent the original parameters but preserving the precision of the original parameters as best as possible.

For more information this example, see [A Visual Guide to Quantization](#).

The Neural Network Processing Assist (NNPA) in IBM Telum II supports 8-bit signed-integer quantized matrix multiplications. The following steps show how to compile an ONNX model for 8-bit quantization on NNPA.

When these steps are not followed, models are still accelerated when targeting the Telum accelerator by using a combination of 16-bit floating-point numbers for computations mapped to the Telum Integrated AI Accelerator and 32-bit floating-point numbers for computations mapped to the Telum CPUs.

Quantization with the IBM zDLC can be used in two ways, depending on the format of the input ONNX model:

1. Quantized input model: If the ONNX model has already been quantized by an external framework (for example, ONNX Runtime), it contains 8-bit operations. In this case, the compiler automatically maps these operations to the appropriate 8-bit instructions for execution on the NNPA. No additional compiler flags are required. This approach supports both static and dynamic quantized models. Because no further compiler-side quantization is needed, this case is not discussed further.
2. Nonquantized input model: If the input model uses high-precision data types (for example, float32), the IBM Z Deep Learning Compiler can apply quantization during compilation.

This process converts the model to a quantized form that is suitable for NNPA execution. In this workflow, the compiler currently supports only dynamic quantization. The remainder of this section focuses on this approach.

Quantization requires NNPA in IBM Telum II. Therefore, the following compile flags must be specified to enable quantization: `-macce1=NNPA -march=z17`.

4.11.1 Dynamic quantization by the compiler

It is important to note that the zDLC compiler supports per-tensor dynamic quantization. It also quantizes data tensors from float32 to 8-bit signed integer. If a data tensor in the input model is already an 8-bit signed integer, the compiler does not quantize it again.

The compiler supports two flags for applying dynamic quantization at compile time:

- ▶ `-nnpa-quant-dynamic`: Enables dynamic quantization.
- ▶ `-nnpa-quant-op-types`: Specifies which ONNX operation types to quantize (for example, MatMul, Conv).

During dynamic quantization, you can control whether activations and weights are quantized symmetrically or asymmetrically by specifying options such as `symActivation`, `asymActivation`, `symWeight`, and `asymWeight` in the `--nnpa-quant-dynamic` flag.

For example, using `--nnpa-quant-dynamic=asymActivation,symWeight` applies asymmetric quantization to activations and symmetric quantization to weights.

If you specify `--nnpa-quant-dynamic` without values, the compiler automatically determines the appropriate quantization options and operation types based on its internal heuristics and default settings.

4.11.2 Performance Notes

Symmetric quantization is often more efficient for inference because it uses simpler arithmetic and has better hardware acceleration support. However, it can sometimes produce slightly reduced accuracy than asymmetric quantization, especially for models with activation distributions that are not centered on zero.

Experiment with both symmetric and asymmetric quantization schemes to determine the best balance between performance and accuracy for your model and deployment scenario.

4.11.3 Limitations

The following list summarizes the current limitations of the IBM zDLC for quantization:

- ▶ Only per-tensor quantization is supported. Scale and zero-point values are computed per tensor and are scalar values.
- ▶ The target quantization data type is an 8-bit signed integer.
- ▶ Asymmetric quantization for weights is not yet supported.

4.12 IBM Z Deep Learning Compiler debug instrumentation

Instrumentation and debugging information can be collected during model runtime by enabling one of several supported methods at compile time by using the IBM zDLC. For detailed instructions and usage examples, refer to the official IBM zDLC documentation available in the product repository:

- ▶ [Profile IR option](#)
- ▶ [Instrument options](#)
- ▶ [NNPAUnsupportedOps option](#)

4.13 IBM zDLC performance features

When compiling models for the IBM Z Integrated Accelerator for AI, the IBM Z Deep Learning Compiler (zDLC) optimizes models to use the accelerator whenever possible. To support a wide range of models, IBM zDLC compiles models so that operators not supported by the accelerator or operators with unsupported settings run on the CPU.

This section outlines performance optimization techniques available when compiling models with the IBM zDLC. It focuses on maximizing runtime efficiency, particularly when targeting the IBM Z Integrated Accelerator for AI (NNPA). The section is divided into three key subsections.

4.13.1 Specifying input tensor dimensions

When running models with multiple dynamic dimensions (for example, input shapes that contain multiple -1 values), use the **--shapeInformation** flag to specify static dimensions and improve runtime performance.

For some models, this flag helps the IBM zDLC determine at compile time which operations are compatible with the accelerator, leading to more efficient execution.

For example, if a vision model has an input tensor with shape (-1, -1, -1, 3) representing (batch, height, width, channels), performance can increase when you specify the height and width dimensions at compile time. To do so, add the following option when compiling the model: **--shapeInformation 0:-1x640x480x3**

If the model has multiple input tensors, those can also be specified using the following option: **--shapeInformation 0:-1x640x480x3,1:-1x100,2:... .**

The **--shapeInformation** flag can be used with **--onnx-op-stats** to determine whether specifying the shape enables more operations to run on the IBM Z Integrated Accelerator for AI. For more information, see [View operation targets at compile time](#).

4.13.2 View operation targets at compile time

The IBM Z Deep Learning Compiler (zDLC) can report, at compile time, the number of operators that run on the CPU versus those that run on the IBM Z Integrated Accelerator for AI.

When compiling the model, add the option `--onnx-op-stats [TXT|JSON]`. Operations that begin with `onnx.*` run on the CPU, and operations that begin with `zhgh.*` run on the IBM Z Integrated Accelerator for AI.

4.13.3 Device placement of operations

When compiling with the IBM Z Deep Learning Compiler (zDLC), each operation is either assigned a target automatically or specified manually. The target determines whether the operation runs on the CPU or on the IBM Z Integrated Accelerator for AI (NNPA).

Specifying a target of NNPA for an operation might not ensure execution on the NNPA. This can occur for the following reasons:

- ▶ The NNPA variant of the operator is not supported.
- ▶ The NNPA does not support the tensor size that is used in the model.
- ▶ The operation runs faster on the CPU.

For details on obtaining or specifying the target for device placement, see [Open source device placement documentation](#).

4.14 IBM Z Deep Learning Compiler scope and versioning

The IBM Z Deep Learning Compiler (zDLC) follows a continuous-delivery model that includes one to two minor releases each year. Bug fixes are included in the next scheduled minor release and are not applied retroactively to earlier versions.

Each release of IBM zDLC is aligned with a specific version of the ONNX-MLIR compiler framework and provides support for both CPU execution and the IBM Z Integrated Accelerator for AI (NNPA). Other ONNX-MLIR-based accelerators are not supported within the scope of IBM zDLC.

For the latest list of supported ONNX operators, operator-set (opset) versions, and any limitations or exclusions, see the related documentation resources. Operators not listed, or those used outside the documented constraints, are considered outside the supported scope of IBM zDLC. For more information, see the following resources:

- ▶ [Supported ONNX Operation for CPU](#)
- ▶ [Supported ONNX Operation for IBM Z Integrated Accelerator \(NNPA\)](#)
- ▶ [Supported Operations for Target NNPA Supplement](#)

Versioning Policy

IBM zDLC follows the [semantic versioning guidelines](#) with a few deviations. These differences account for IBM zDLC as a compiler and are outlined in the following sections. Each zDLC release follows semantic versioning in the format `major.minor.patch`.

Major / Version

All releases with the same major number have runtime APIs that are compatible with earlier releases. That means programs designed to run model libraries generated by IBM zDLC X.0 can run model libraries generated by later IBM zDLC releases that have the same major version (for example, X.0, X.1, X.2, and so on). Programs that are designed to run model libraries that are generated by IBM zDLC X.1 can run model libraries that are generated by IBM zDLC X.1, X.2, and so on.

Changes in major releases can include one or more of the following updates:

- ▶ Changes to runtime APIs that are incompatible with earlier releases. Programs that import model libraries generated by the IBM Z Deep Learning Compiler (zDLC) might require updates between major releases, depending on the affected runtime API language.
- ▶ Changes to critical or general model compilation flags that are incompatible with earlier releases.
- ▶ Significant feature additions that extend beyond normal minor release updates.

Note: Prebuilt pybind11 PyRuntimes for versions of Python that have reached the end of life can be removed without a major release increase.

Minor / Feature

Minor releases typically include new features, enhancements, and bug fixes.

IBM strives to keep minor releases fully compatible with earlier versions. However, there might be cases where performance, debug, or accelerator-specific compilation flags change in ways that are not compatible with earlier releases.

Support for end-of-life languages can be removed in minor releases.

Patch / Maintenance

Patch releases contain only bug fixes, security updates, or updates to non-IBM zDLC packages included in the IBM zDLC containers. IBM zDLC introduces only compatible changes in patch updates.



IBM Snap ML

IBM Snap Machine Learning, commonly referred to as Snap ML, is a high-performance machine learning library designed for large-scale, low-latency workloads on IBM Z systems. It is engineered to address key challenges in classical machine learning pipelines:

- ▶ Slow training on large datasets
- ▶ Time-consuming pre-processing that bottlenecks the entire pipeline
- ▶ Inference latency that impacts real-time applications

These challenges are especially critical for enterprises running ML algorithms on large datasets where throughput and responsiveness are essential.

Snap ML accelerates training and inference of ML models by providing high-performance implementations of Generalized Linear Models, tree-based models, and Gradient Boosting Machine models.

For the latest features, updates, and documentation, see the [Snap ML documentation](#).

This chapter includes the following topics:

- ▶ 5.1, “Unlocking the full potential of machine learning with IBM Snap ML” on page 52
- ▶ 5.2, “Graph feature preprocessor” on page 57
- ▶ 5.3, “C++ preprocessing pipeline” on page 58
- ▶ 5.4, “Model import functionality:” on page 61
- ▶ 5.5, “Installation guide” on page 62
- ▶ 5.6, “Snap ML End to End Model Deployment” on page 62

5.1 Unlocking the full potential of machine learning with IBM Snap ML

To fully harness the power of machine learning in enterprise environments, especially those that require real-time decision-making, and high-throughput inference, organizations must use hardware-accelerated solutions. IBM Snap ML is designed with this goal in mind, offering a suite of performance-optimized tools that integrate seamlessly with IBM Z.

At the core of this innovation is the IBM Integrated Accelerator for AI, a specialized component built into the IBM Telum processor. This accelerator enables Snap ML to deliver high-speed inference for complex models, particularly decision-tree ensembles, by transforming traditional algorithms into tensor-based operations. IBM Snap ML uses this hardware to achieve efficient and scalable inference for enterprise AI workloads.

5.1.1 Efficient inference by using IBM Integrated AI Accelerator

IBM Telum is the next-generation processor for IBM z16 and later systems. The Telum processor features a dedicated on-chip accelerator that provides real-time insights into transactional data as it is being processed.

The IBM Integrated Accelerator for AI is designed to efficiently run computations common to deep neural-network inference, including matrix-matrix multiplications and convolutions. However, inference of decision-tree ensembles involves traversing trees that contain conditional operators and nonuniform memory access patterns.

IBM Snap ML uses a tensor-based machine-learning inference algorithm that transforms tree traversal into a set of tensor operations. These operations can be offloaded to the Integrated Accelerator for AI on IBM Z for fast execution. Decision-tree-based ensemble models, such as random forests and boosting machines, deliver significant runtime performance advantages when using accelerator-based inference¹.

The Snap ML library provides a Python API for model import:

```
import_model(input_file, input_type, tree_format='auto', X=None)
```

If you set the **tree_format** parameter to **zdnntensors**, the model is optimized to run on the Integrated AI Accelerator on IBM Z. This optimization uses a tensor-based inference algorithm designed for high-performance execution.

By default, **tree_format** is set to **auto**. In this mode, Snap ML checks for the availability of the Integrated AI Accelerator on IBM z16 or IBM z17. If the accelerator is available, the model is optimized by using **zdnntensor**, which uses a tensor-based inference algorithm for high-performance execution. If the accelerator is not available, the model is optimized by using **compress_trees**, which targets CPU execution through the Snap ML compressed decision-tree inference approach.

5.1.2 Model-agnostic inference pipeline

A key feature of IBM Snap ML is its model-independent inference pipeline. This pipeline enables you to integrate and accelerate tree-based models trained in popular machine-learning frameworks, such as scikit-learn, XGBoost, and LightGBM, for deployment in production environments.

¹ <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&number=10181908>

Deploy these models by completing the following steps:

1. Export the trained model to a format supported by Snap ML, such as PMML, JSON, or ONNX.
2. Import and transform the model into Snap ML's optimized internal representation.
3. Run fast predictions on either the CPU or the IBM Z Integrated AI Accelerator.

Figure 5-1 illustrates the Snap ML model-independent inference pipeline. It shows how models trained in various frameworks, such as scikit-learn, XGBoost, LightGBM, and Snap ML, can be exported to supported formats such as PMML, JSON, or ONNX. These models are then imported and transformed into Snap ML's internal structures, either as compressed trees or tensor-based trees.

The scoring server automatically selects the optimal inference method based on system capabilities and model characteristics. The Snap ML prediction engine runs fast, hardware-optimized inference by using either the CPU or the IBM Z Integrated AI Accelerator.

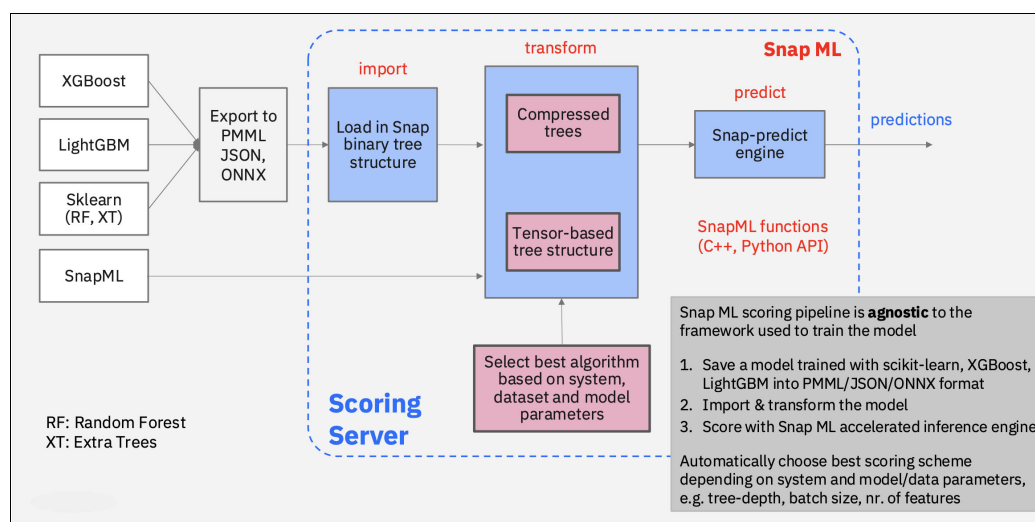


Figure 5-1 Model-independent Snap ML scoring pipeline

5.1.3 Supported model types and formats

IBM Snap ML supports the import of tree-ensemble models that are trained with popular machine-learning frameworks such as scikit-learn, XGBoost, and LightGBM. You can export these models in supported formats (PMML, JSON, or ONNX) and import them into Snap ML for accelerated inference.

Details about which Snap ML classes support specific model types and formats are available in the [Snap ML model import documentation](#).

Internally, Snap ML uses a tensor-based tree structure to enable fast decision-making. It applies specialized optimization schemes that are tailored to different types of tree models. The following tree-model types are supported:

- ▶ Random Forests (RF)
- ▶ Extra Trees (XT)
- ▶ Boosted Trees (XGBoost, LightGBM)

Snap ML provides a Python API that closely mirrors the scikit-learn interface. By using this design you can migrate existing workflows with minimal code changes while benefiting from Snap ML performance optimizations.

Figure 5-2 compares the syntax for training a Random Forest classifier by using scikit-learn and Snap ML. The Snap ML API closely mirrors the scikit-learn interface, enabling a smoother transition for users. In addition to standard parameters, Snap ML introduces advanced configuration options (such as `use_gpu`, `gpu_ids`, and `use_histograms`) to support hardware acceleration and histogram-based optimization. These enhancements deliver scalable performance while maintaining ease of use for data scientists familiar with scikit-learn.

Scikit-learn RandomForest API	Snap ML RandomForest API
<pre> from sklearn.ensemble import RandomForestClassifier as SklearnForest rf = SklearnForest(random_state=42, max_depth=10, n_estimators=100, n_jobs=8, max_features='sqrt') #Train rf.fit(X_train, y_train) #Inference pred_test = rf.predict(X_test) </pre>	<pre> from snapml import RandomForestClassifier as SnapForest rf = SnapForest(random_state=42, max_depth=10, n_estimators=100, n_jobs=8, max_features='sqrt') #Train rf.fit(X_train, y_train) #Inference pred_test = rf.predict(X_test) </pre>

Figure 5-2 Scikit-learn vs Snap ML RandomForest API

Figure 5-3 illustrates the syntactic similarity between the scikit-learn and IBM Snap ML APIs for training and using tree-based classifiers, such as Random Forest and Gradient Boosting. This side-by-side comparison shows how Snap ML preserves the familiar structure of scikit-learn while introducing additional parameters (such as `use_gpu`, `gpu_ids`, and `use_histograms`) and depth constraints (such as `min_max_depth`) to support hardware acceleration and performance tuning.

Designed for high-performance inference, especially on accelerators such as IBM Telum and Telum II, you can use Snap ML to transition existing workflows with minimal code changes while gaining access to advanced optimization features for low-latency and high-throughput environments.

XGBoost scikit-learn API	SnapBoost API
<pre> from xgboost import XGBClassifier as Xgb booster = Xgb(n_estimators=1000, max_depth=8, learning_rate=0.01, n_jobs=8) #Train booster.fit(X_train, y_train) #Inference yhat_test = booster.predict(X_test) </pre>	<pre> from snapml import BoostingMachineClassifier as Snapboost booster = Snapboost(n_estimators=1000, min_max_depth=8, max_min_depth=8, learning_rate=0.01, n_jobs=8) #Train booster.fit(X_train, y_train) #Inference yhat_test = booster.predict(X_test) </pre>

Figure 5-3 XGBoost vs SnapBoost API

In summary, Figure 5-2 on page 54 compares the training and inference syntax of a Random Forest classifier by using scikit-learn and IBM Snap ML, and Figure 5-3 on page 54 shows the compatibility between the Snap ML API and the XGBoost library.

Example 5-1 provides an overview of the machine-learning models available in the Snap ML library. The models are organized by model type into three main categories. These categories include the following types of models:

- ▶ Classification and regression models
- ▶ Tree-based models
- ▶ Ensemble models

This categorization helps you quickly identify the appropriate Snap ML class for your specific machine-learning task, whether you are building linear models, decision trees, or advanced ensemble methods.

Example 5-1 Python supportedclasses

```
# Classification and Regression Models
snapml.LogisticRegression
snapml.LinearRegression
snapml.SupportVectorMachine

# Tree-based Models
snapml.DecisionTreeClassifier
snapml.DecisionTreeRegressor
snapml.RandomForestClassifier
snapml.RandomForestRegressor

# Ensemble Models
snapml.BoostingMachineClassifier
snapml.BoostingMachineRegressor
snapml.BatchedTreeEnsembleClassifier
```

5.1.4 Multi-output calibrated classifier

IBM Snap ML includes the [MultiOutputCalibratedClassifier](#), which enables probability calibration for multi-output (multi-label) classification tasks. This functionality is especially useful when working with models that do not natively support multi-target classification, allowing you to obtain calibrated probability estimates for each label independently.

Using `MultiOutputCalibratedClassifier` has several key benefits:

- ▶ Probabilistic interpretation of predictions
- ▶ Parallel training and inference by using the `n_jobs` parameter
- ▶ Seamless integration with Snap ML classifiers or scikit-learn classifiers

Example 5-2 on page 56 demonstrates how to build a multi-label classification pipeline by using a calibrated base classifier in Python. This approach is useful when you need to generate well-calibrated probability estimates for each label in a multi-output classification task. The workflow wraps a calibrated binary classifier with a `MultiOutputClassifier`, enabling parallel training and inference across multiple targets.

Example 5-2 Snap ML MultiOutput Calibrated Classifier Sample Usage

```
# Step 1: Prepare input features (X) and output targets (y)

# Step 2: Define base classifier
base_model = SomeBinaryClassifier(...)

# Step 3: Calibrate the base classifier
calibrator_model = CalibratedClassifierCV(base_estimator=base_model,
method='sigmoid', cv=5)

# Step 4: Wrap with MultiOutputClassifier
multi_output_model = MultiOutputClassifier(estimator=calibrator_model, n_jobs=-1)

# Step 5: Fit model on training data
multi_output_model.fit(X_train, y_train)
```

5.1.5 Multi-Threaded Inference in Snap ML

IBM Snap ML provides native support for multi-threaded inference, enabling you to fully use the multi-core architecture of IBM Z systems. This capability improves performance by allowing parallel execution across available processor cores.

To enable multi-threading during inference, set the number of threads by using the model parameter `n_jobs`. This configuration helps scale inference workloads efficiently, especially in high-throughput environments. The `n_jobs` parameter controls how many threads Snap ML uses internally for parallel processing. If `n_jobs` is not set, Snap ML defaults to using a single thread.

Example 5-3 demonstrates how to import a previously trained Random Forest regression model into Snap ML by using the `import_model` method. This workflow shows how to load a model in PMML format, configure multi-threaded inference, and generate predictions by using the Snap ML `RandomForestRegressor` class. This approach enables efficient deployment of pre-trained models with minimal code changes.

Example 5-3 Snap ML Multi threading configuration

```
from snapml import RandomForestRegressor

# Load a previously exported model (PMML format)
model = RandomForestRegressor()
model.import_model("model.pmml", input_type="pmml")

# Enable multi-threading with 4 threads
model.set_params(n_jobs=4)
print("Using %d threads for inference" % model.n_jobs)

# Perform inference
predictions = model.predict(X)
```

5.1.6 Enabling verbose logging in Snap ML

By default, IBM Snap ML is designed for efficient and quiet execution. It emits log messages only to standard output or error streams and remains silent unless an error occurs. If you need to monitor training progress or collect detailed iteration-level statistics, enable verbose mode.

To activate verbose output during model training, set the parameter `verbose=True` when initializing the model. This option is useful for debugging and performance tuning in iterative training workflows.

Example 5-4 demonstrates how to train a gradient-boosting classifier by using the `BoostingMachineClassifier` in Snap ML. This workflow shows how to load input data, configure the classifier with key parameters, and start training. The verbose flag enables per-iteration logging, which is helpful for tracking training progress. This example illustrates the ease of use of Snap ML and its performance-oriented design for scalable machine-learning tasks.

Example 5-4 Enabling Verbose Logging in Snap ML

```
import numpy as np
from snapml import BoostingMachineClassifier

x = np.load("/data/X_bc.npy")
y = np.load("/data/y_bc.npy")

clf = BoostingMachineClassifier(
    num_round=10,
    verbose=True, # Enables per-iteration logging
)

print("Starting training classifier.")
clf.fit(X, y)
print("Finished training classifier.")
```

5.2 Graph feature preprocessor

The IBM Snap ML `GraphFeaturePreprocessor` (GFP) extracts graph-based features from transaction or event data that you represent as edge lists. The GFP transforms temporal graphs into feature-rich inputs for machine-learning models by identifying structural patterns, temporal behaviors, and connectivity dynamics.

The `GraphFeaturePreprocessor` supports large-scale graphs by using multi-threading and temporal memory control. Temporal memory control enables the preprocessor to manage how long timestamped edges remain in memory, allowing it to focus on the most relevant time-based patterns. This approach improves scalability and ensures that feature extraction reflects recent and meaningful graph activity. The preprocessor returns enriched feature arrays that you can use directly to train models for fraud detection, credit scoring, and other graph-based classification tasks.

Example 5-5 on page 58 demonstrates how to use the `GraphFeaturePreprocessor` class from the Snap ML library to extract graph-based features from edge data. This preprocessor enriches edge representations by computing structural and temporal features that can improve downstream machine-learning models. The example shows how to configure the

preprocessor, provide an edge list, and generate feature-enhanced output by using the `fit_transform` method.

Example 5-5 Using the `GraphFeaturePreprocessor` in Snap ML

```
from snapml import GraphFeaturePreprocessor
import numpy as np

# Initialize and configure the preprocessor
gfp = GraphFeaturePreprocessor()
gfp.set_params({'is': True, 'vertex_stats': True, 'time_window': 10})

# Provide an edge list (Edge ID, Src, Dst, Timestamp...)
edge_list = np.array([
    [1001, 1, 2, 10, 50],
    [1002, 2, 3, 20, 70],
    # Additional edges...
])

# Fit and transform: builds graph and returns feature-enriched edges
features_out = gfp.fit_transform(edge_list)
```

See the documentation for [Snap ML `GraphFeaturePreprocessor`](#) for more details.

5.3 C++ preprocessing pipeline

Other than accelerated training and inference, IBM Snap ML includes a suite of high-performance preprocessing transformers implemented in C++. These transformers eliminate performance bottlenecks in the data-preparation pipeline, enabling efficient end-to-end machine-learning workflows.

5.3.1 Exporting a preprocessing pipeline

You can build preprocessing pipelines by using the familiar scikit-learn interface in Python. With a single API call, you can export the entire pipeline to a standardized JSON format without modifying your existing code. This JSON representation captures the full preprocessing workflow, including all transformations, parameters, and their execution order, in a portable, language-agnostic format.

The exported pipeline can be consumed directly by the IBM Triton Inference Server, which uses the Snap ML C++ backend to run the complete inference pipeline efficiently.

Snap ML offers the following high-performance transformers:

- ▶ **KBinsDiscretizer**: Bins continuous features into discrete intervals
- ▶ **Normalizer**: Scales individual samples to unit norm
- ▶ **OneHotEncoder**: Encodes categorical features as binary vectors
- ▶ **OrdinalEncoder**: Encodes categorical features as ordinal integers
- ▶ **FunctionTransformer**: Applies custom functions to features
- ▶ **TargetEncoder**: Encodes categorical features by using target statistics

By implementing these transformers in optimized C++ rather than Python, Snap ML removes critical delays from the inference pipeline, making it suitable for high-throughput and low-latency production environments.

5.3.2 Using a preprocessing pipeline in Snap ML

You can use IBM Snap ML to define preprocessing pipelines by using the familiar scikit-learn interface in Python. These pipelines can include a sequence of transformations, such as encoding, normalization, or discretization, that are applied to raw input data before model inference.

After you define the pipeline, you can export it to a standardized JSON format by using Snap ML's API. This JSON representation captures the complete preprocessing logic, including transformer types, parameters, and execution order. You can then deploy the exported pipeline to the IBM Triton Inference Server, where the Snap ML C++ backend efficiently executes the preprocessing steps alongside model inference.

You can export a preprocessing pipeline created in scikit-learn to JSON format without modifying it. Later, you can use this export format during model serving on the Triton Inference Server C++ backend, which is discussed in a later chapter.

Before exporting the pipeline, fit it on training data. This step enables the pipeline to learn encodings and bins. After fitting, you can export the preprocessing logic to JSON.

Example 5-6 demonstrates how to export a trained preprocessing pipeline from Snap ML to a standardized JSON format. This workflow shows how to define and fit a pipeline by using the scikit-learn interface, extract the preprocessor component, and serialize it for deployment. The exported JSON captures the entire preprocessing logic, including transformer types, parameters, and sequence, and can be consumed by the IBM Triton Inference Server for efficient execution by using the Snap ML C++ backend.

Example 5-6 Exporting a Snap ML preprocessing pipeline to JSON for deployment

```
from snapml import export_preprocessing_pipeline

# Fit your preprocessing pipeline on training data
preprocess_pipeline.fit(X_train, y_train)

# Extract the preprocessor component
preprocessor = preprocess_pipeline.named_steps['preprocessor']

# Export to JSON format
export_preprocessing_pipeline(preprocessor, 'pipeline.json')
```

This file can be used during model inference, even in a different environment, to apply the exact same preprocessing.

Note: You can use the preprocessing pipeline for end-to-end inference on the Triton Inference Server Snap ML C++ backend. Deploy your model on this backend to take advantage of this feature.

5.3.3 Understanding pipeline.json

The pipeline.json file defines the structure and logic of a preprocessing pipeline used by IBM Snap ML on the IBM Triton Inference Server C++ backend. It contains two main sections:

1. data_schema

The data_schema section specifies how Snap ML interprets each column in the input data. It includes the following keys:

- num_indices: Lists the column positions that contain numerical data.
- cat_indices: Lists the column positions that contain categorical data.

Important: Do not shuffle or change the order of columns after exporting pipeline.json. The input data used during inference must match the exact column order used when exporting the pipeline. Any mismatch results in incorrect preprocessing.

2. transformers

The transformers section defines the transformations applied to each group of columns before the data is passed to the model. Each transformer entry includes the following information:

- The type of transformation (for example, Normalizer, KBinsDiscretizer)
- The parameters and data used by the transformer
- The columns to which the transformation applies

The transformers section acts as an instruction guide for preprocessing, ensuring that the input data is transformed consistently and accurately before inference.

Example 5-7 is a JSON snippet that illustrates how Snap ML represents a preprocessing pipeline for deployment on the IBM Triton Inference Server. The structure defines a sequence of transformers, each applied to specific input columns. The configuration includes transformer types, parameters, and the columns they operate on, enabling consistent preprocessing during model inference.

Example 5-7 Sample pipeline.json structure

```
{
  "transformers": {
    "transformer1": [
      {
        "type": "Normalizer",
        "params": { ... },
        "data": { ... },
        "columns": [0, 1, 2]
      }
    ],
    "transformer2": [
      {
        "type": "KBinsDiscretizer",
        "params": { ... },
        "data": { ... },
        "columns": [3, 4, 5]
      }
    ]
  }
}
```

The number of transformers in the pipeline.json file depends on how many preprocessing layers you define in your pipeline. Each transformer entry includes the following fields:

- ▶ type: Specifies the kind of transformation applied (for example, Normalizer, KBinsDiscretizer, OneHotEncoder)
- ▶ params: Contains the settings or hyperparameters used to configure the transformer
- ▶ data: Stores the fitted parameters that the transformer learned during training as in the following examples:

- A KBinsDiscretizer includes bin_edges.
- A OneHotEncoder includes categories.
- columns: Lists the column indexes that the transformer operates on.

Each transformer ensures that the input data is processed consistently and accurately before it is passed to the model for inference.

Figure 5-4 demonstrates how the preprocessing pipeline is used during inference on the IBM Triton Inference Server Snap ML C++ backend. This setup provides efficient, end-to-end model serving by combining preprocessing and inference in a single runtime environment.

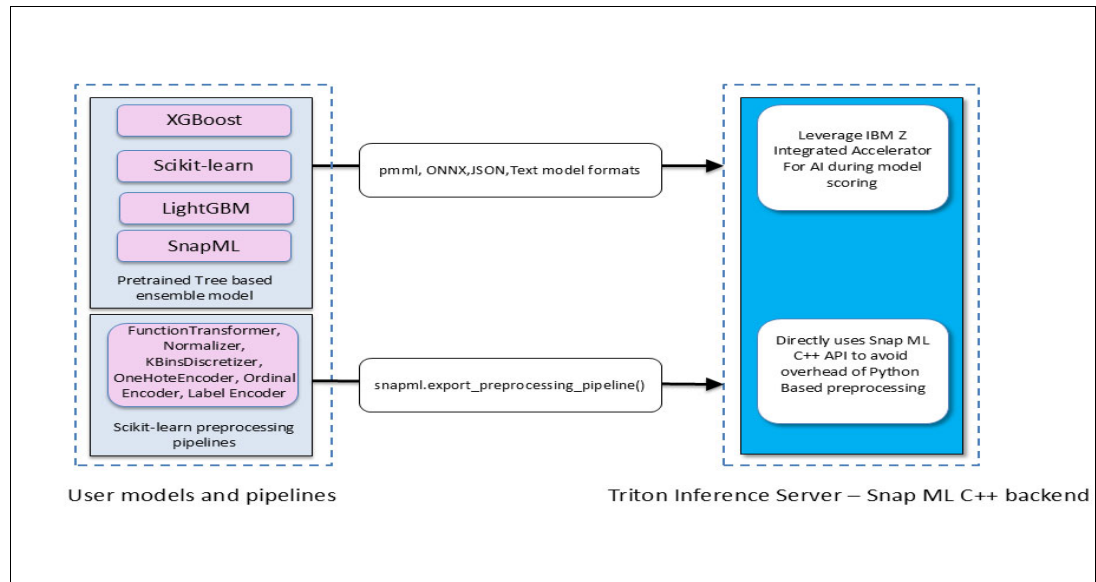


Figure 5-4 End-to-end pipeline on Triton Inference Server Snap ML C++ backend

For a complete end-to-end preprocessing example and a sample pipeline.json, see the following website:

<https://github.com/IBM/ibmz-accelerated-for-nvidia-triton-inference-server/tree/main/examples/triton-credit-card-fraud-detection/ibmsnapml-backend>

5.4 Model import functionality:

Snap ML supports importing tree ensemble models trained with other frameworks, so you can take advantage of the Snap ML accelerated inference engine, regardless of the original training environment.

5.4.1 Import Methods

You can import a model using one of the following approaches:

- Class-specific import: Instantiate the corresponding Snap ML class and call the `import_model()` method.
- Generic import: Use the `snapml.import_model()` function, which automatically detects the model type.

5.4.2 Platform Compatibility Considerations

Snap ML models saved using pickle or joblib are platform-dependent because of binary serialization. Specifically, you cannot save a model on an Intel (x86_64) platform and load it on an IBM Z (s390x) platform.

To ensure cross-platform compatibility, use the PMML (Predictive Model Markup Language) format. PMML provides a platform-independent way to serialize and deploy models.

You can generate a PMML file by running the following method in your code to create the file:

```
model.export_model('model.pmm1')
```

5.5 Installation guide

IBM Snap ML is available as part of Machine Learning for z/OS, and for IBM LinuxONE and Linux on Z environments, including z/OS container extensions. A prebuilt container is available as part of the IBM AI Toolkit Offering. Also, Snap ML can be installed by using PyPI, and Snap ML is a component of Cloud Pak for Data on Linux on Z.

For more information and details, see [SnapML Installation Guide](#) and [Using the IBM Z Accelerated for Snap ML Container Image](#).

5.6 Snap ML End to End Model Deployment

An end-to-end model deployment that demonstrates a credit card fraud detection use case can be efficiently deployed by using the Snap ML high-performance features can be found at the following website:

<https://github.com/IBM/ibmz-accelerated-for-nvidia-triton-inference-server/tree/main/examples/triton-credit-card-fraud-detection/ibmsnapml-backend>



IBM Z Accelerated for NVIDIA Triton Inference Server

IBM Z supports AI workloads that range from traditional transactional processing to complex, data-intensive operations. Modern enterprise applications require a variety of AI capabilities: some use machine learning for statistical analysis, others rely on deep learning, and many combine both approaches.

Meeting these diverse requirements at enterprise scale demands inference servers that handle live data streams, adapt to evolving models, and deliver real-time responses with the reliability and security required for mission-critical applications. Handling live data, adapting to evolving models, and delivering real-time responses are key requirements for enterprise inference servers.

The Triton Inference Server on IBM Z supports training AI models anywhere and deploying them efficiently on IBM Z.

This chapter describes the architecture of Triton, which provides low latency and high throughput for inference workloads. It also outlines backend support, including the Python backend and IBM custom backends. The chapter guides you through preparing trained models, optimizing inference performance, and using Triton Inference Server features on IBM Z during deployment.

This chapter includes the following topics:

- ▶ 6.1, “Train anywhere and deploy on IBM Z by using Triton Inference Server” on page 64
- ▶ 6.2, “Deploying strategies in production” on page 69
- ▶ 6.3, “Sample usage and deployment options” on page 71
- ▶ 6.4, “End-to-end model deployment use cases with IBM TIS” on page 73
- ▶ 6.5, “IBM Z accelerated for TIS: Best practices” on page 73

6.1 Train anywhere and deploy on IBM Z by using Triton Inference Server

IBM Z with Triton Inference Server supports the scalable deployment of models trained on any platform. This flexibility helps you avoid costly and time-consuming model retraining because you can reuse existing AI models.

The following sections describe how Triton Inference Server and its architecture on IBM Z support production deployment features such as diverse AI framework support, scalable model serving, IBM custom backends, version control, dynamic batching, and maintainable APIs.

6.1.1 Triton Inference Server architecture on IBM Z

The Triton Inference Server architecture on IBM Z is designed to support the z/Architecture big-endian format while delivering optimal performance for AI workloads. This section describes the architectural components that enable this support.

Figure 6-1 illustrates the architecture of IBM Z Accelerated for NVIDIA Triton Inference Server. The diagram shows how client requests that are received through HTTP or gRPC are processed by dynamic batching and model-specific scheduler queues before being routed to supported backends, including Python, IBM Snap ML C++, IBM ONNX-MLIR, and IBM PyTorch, all running on IBM Z.

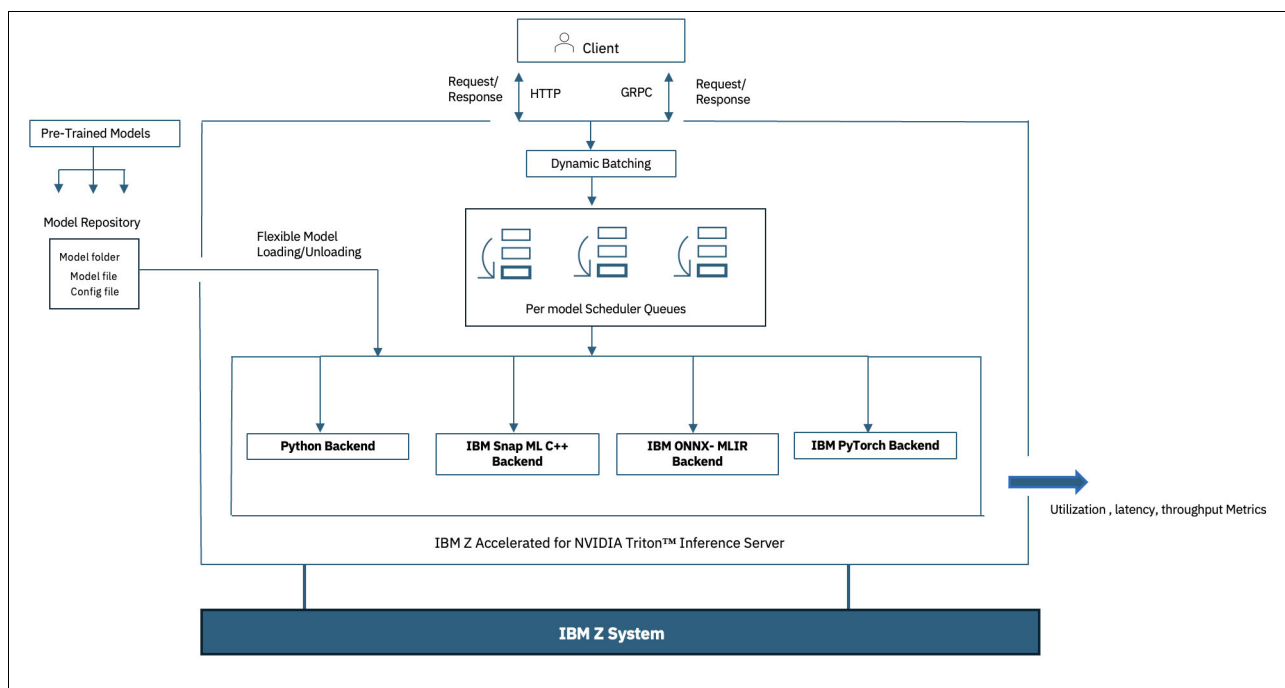


Figure 6-1 IBM Z Accelerated for NVIDIA Triton Inference Server Architecture

For more information about the Triton Inference Server architecture, see [Triton Architecture](#).

6.1.2 Understanding Triton Inference Server components on IBM Z

The Triton Inference Server on IBM Z is composed of several key components that work together to support scalable and efficient AI model deployment. This section describes the following components:

- ▶ Backend
- ▶ Model repository
- ▶ Schedulers

Backends

The Triton Inference Server backend on IBM Z runs inference by using supported frameworks such as Python, IBM Snap ML C++, IBM ONNX-MLIR, and IBM PyTorch. A backend acts as the intermediary between the AI model and Triton's execution engine, handling input data, scheduling model execution, and returning inference results. You can find the available backends that are supported by Triton on IBM Z under the path `/opt/tritonserver/backends` inside the container.

The model repository stores trained models in a structured format, enabling version control and simplified updates. Schedulers manage inference requests by organizing them into queues and applying techniques such as dynamic batching to optimize performance.

For more details on Triton backends, see <https://github.com/triton-inference-server/backend>.

In addition to supporting native Triton Inference Server backends such as Python, IBM Z provides customized backends optimized for its architecture. These backends are specifically designed to deliver high throughput, low latency, and IBM-specific features that enhance usability for AI workloads. Each backend takes advantage of both IBM Z CPUs and IBM Telum processors to maximize performance.

The following sections introduce the available IBM customized backends and explain how they integrate with Triton to support efficient and scalable inference workflows.

IBM Snap ML C++ Backend

The IBM Snap ML C++ Backend is designed for high-performance machine-learning inference and is optimized for traditional machine-learning workloads. It supports end-to-end inference, including preprocessing steps such as normalization, binning, and one-hot encoding, as described in the Snap ML chapter.

These preprocessing components can be serialized into a `pipeline.json` file by using Snap ML APIs. The components can be integrated directly into the Triton Inference Server workflow, eliminating the need for external preprocessing services.

The IBM Snap ML C++ Backend includes several key features:

- ▶ Optimized C++ implementation for minimal latency
- ▶ Full support for traditional machine-learning models, including tree ensembles, linear regression, logistic regression, and gradient boosting
- ▶ Hardware-aware execution that uses IBM Z integrated on-chip AI acceleration
- ▶ Built-in support for Snap ML preprocessing pipelines
- ▶ End-to-end inference with optional probabilistic output support
- ▶ Seamless integration with models trained in frameworks such as scikit-learn, XGBoost, and LightGBM

- Configurable CPU thread allocation per inference call to enable parallel execution and improve performance

IBM PyTorch Backend

The IBM PyTorch backend is built for IBM Z systems and uses the IBM Integrated Accelerator for AI (NNPA) and the IBM Z Spyre AI Chip through the `ibmz-accelerated-for-pytorch` framework. This backend enables efficient deployment of PyTorch models with minimal integration overhead.

The IBM PyTorch backend includes the following advanced capabilities:

- Support for custom PyTorch models defined in a `model.py` file that extends `torch.nn.Module`, and associated weights (`model.pth`, `model.bin`) containing the `model.state_dict`
- Modular weight loading with flexible path resolution
- Native Hugging Face checkpoint loading by using `from_pretrained`
- Plug-and-play interface with (no script conversion or manual tokenization required)
- Deployment of encoder-based Hugging Face models by using only trained weights

The backend accepts multiple input types, including `INT`, `FLOAT`, and `STRING`. For Hugging Face checkpoints, it supports both pre-tokenized and raw string inputs, with on-the-fly tokenization enabled through `tokenizer_instance_name` and `tokenizer_kwargs`.

To enhance runtime performance, the backend includes model warmup to ensure consistent first-request latency by preloading and transforming weights for NNPA. It also supports hybrid execution by using both the CPU and NNPA to optimize inference throughput.

IBM ONNXMLIR-Triton Backend

The IBM ONNX-MLIR Triton backend enables optimized execution of ONNX models on IBM Z by using the capabilities of ONNX-MLIR and the IBM Z Deep Learning Compiler (zDLC), as described in Chapter 4, “IBM Z Deep Learning Compiler” on page 25. This backend is designed to target the IBM Z architecture and is optimized for low-level execution on the IBM Z AI Accelerator (zAIU).

The backend uses shared object libraries (`model.so`) to enable high-performance inference and takes advantage of low-level instruction sets that are tailored for IBM Z systems.

Through integration with zDLC, ONNX models are automatically converted into optimized `.so` files. These compiled models benefit from hardware-specific optimizations, support for complex model topologies, and compatibility with existing ONNX toolchains. A raw `.onnx` model can be compiled by using zDLC, and the resulting `model.so` file can be deployed directly by using the ONNX-MLIR backend in the Triton Inference Server.

Model Repository

The Triton Inference Server uses a model repository to store trained models in a structured format. This organization enables version control and simplifies model updates. The server automatically discovers and loads models from the repository during startup or when models are requested dynamically.

This repository structure supports efficient deployment workflows and ensures consistency across inference sessions.

Figure 6-2 on page 67 illustrates the architecture of IBM Z Accelerated for NVIDIA Triton Inference Server. The diagram shows how client requests are processed through dynamic

batching and model-specific scheduler queues before being routed to supported backends such as Python, IBM Snap ML C++, IBM ONNX-MLIR, and IBM PyTorch, using models stored in a centralized repository.

```
model_repository/  
└─ my_model/                               # Name of the model  
    └─ config.pbtxt                         # Model configuration file  
        └─ 1/                             # Model version directory (version 1)  
            └─ model_file                  # Model file (e.g., model.onnx, model.pt, model.pmmml, etc.)
```

Figure 6-2 Triton Model Repository Structure

The repository includes the following key principles:

- ▶ The model directory name must match the model name specified in the configuration.
- ▶ Numeric subdirectories represent model versions so that multiple versions can coexist.
- ▶ The configuration file (`config.pbtxt`) defines model metadata, inputs, outputs, and execution parameters.

Triton automatically discovers and loads models from the repository during server startup or when models are dynamically requested. This structure enables efficient and consistent model deployment.

For more information, see [Triton model repository](#).

Model version control

Triton Inference Server supports multiple versions of the same model within a structured repository, which enables the following features:

- ▶ Safe rollbacks: Instantly revert to a previous model version if any issue persists
- ▶ Deployment testing: Run multiple versions of model versions in parallel to test new models on real traffic with minimal risk
- ▶ Zero downtime: New versions can be loaded dynamically without restarting the Triton Inference server

Triton's model management features also support background loading of new models, allowing uninterrupted inference on currently deployed versions. These capabilities help reduce service downtime and streamline production deployment workflows.

For more information, see [Triton model management](#).

6.1.3 Schedulers and running models concurrently

Schedulers

Schedulers play a central role in orchestrating efficient inference within Triton Inference Server. They manage incoming inference requests by organizing them into queues and applying techniques such as dynamic batching and concurrent model execution to optimize throughput and latency.

By controlling how and when inference requests are executed, schedulers help ensure efficient resource usage. Without proper scheduling, the server might experience increased

latency or dropped requests. Triton's scheduling capabilities support scalable, reliable inference across diverse workloads.

For more information, see [Triton Schedulers](#).

Running models concurrently

The Triton Inference Server supports running concurrent models to handle high volumes of inference requests efficiently. It allows multiple instances of the same model to run in parallel, with each instance processing requests independently.

When multiple requests for the same model arrive simultaneously, Triton distributes them across the available model instances based on the configured instance count. This distribution prevents any single instance from becoming a bottleneck and helps maintain scalable performance, even during usage spikes.

Figure 6-3 illustrates the architecture of IBM Z Accelerated for NVIDIA Triton Inference Server. The diagram shows how client requests are processed through dynamic batching and model-specific scheduler queues before being routed to various backends running on IBM Z. Performance metrics such as utilization, latency, and throughput are highlighted to demonstrate system efficiency.

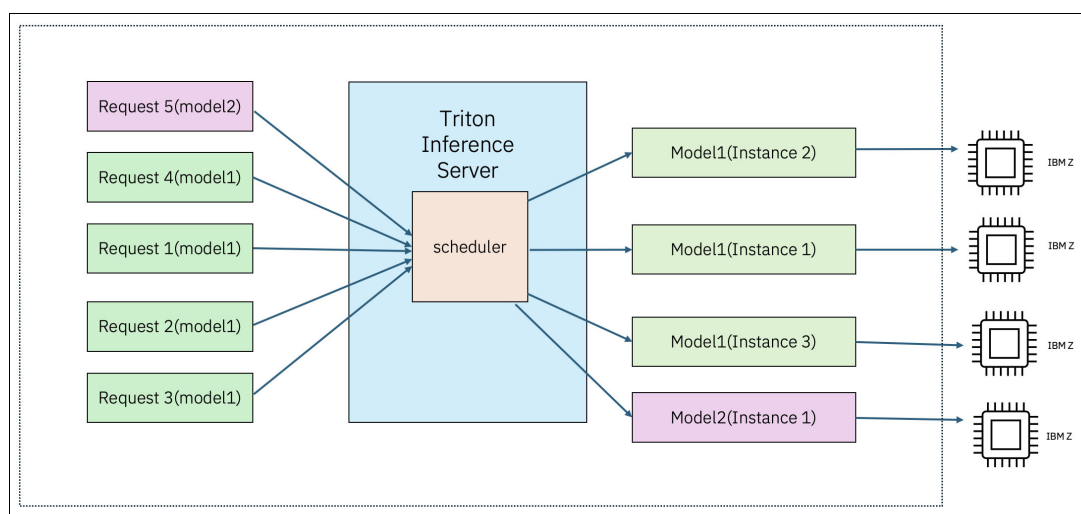


Figure 6-3 Concurrent model execution

Running models concurrently enables efficient use of system resources by allowing multiple inference requests to be processed simultaneously. Three key mechanisms contribute to this capability:

1. Instance management

In the `config.pbtxt` file, you can specify the number of model instances by using the `instance_group` parameter. Based on the configured instances, the scheduler automatically distributes the inference load across them. Each model instance loads in parallel during server startup, enabling faster readiness and improved concurrency. For more information about `instance_group`, see [Instance Groups](#).

2. Multi-core usage

Multi-core usage enables optimal performance by distributing workloads across multiple CPU cores. IBM Z is designed to take advantage of this architecture, allowing thread-safe execution with minimal overhead. If you configure the model with n instances, the first n inference requests are run in parallel. Any additional requests wait until one of the running

workload completes, ensuring efficient use of available cores without overloading the system.

3. Dynamic batching

Dynamic batching is a feature of the Triton Inference Server that automatically combines multiple incoming inference requests into a single batch. This process requires no changes on the client side. The server processes the batched input in the same way that it handles a standard training batch, enabling efficient execution.

Dynamic batching reduces the number of CPU or AI accelerator invocations, which improves hardware usage and allows more inferences to be processed in less time. It also lowers the per-request processing time, which reduces latency for real-time applications.

For more information about dynamic batching, see [Dynamic Batching & Concurrent Model Execution](#).

6.2 Deploying strategies in production

This section outlines the key components involved in deploying the Triton Inference Server in a production environment on IBM Z systems.

6.2.1 Containerized deployment

The IBM Z Triton Inference Server is available as a containerized solution that enables flexible deployment and streamlined integration across IBM Z environments. The container image is fully compatible with IBM Z and adheres to Open Container Initiative (OCI) standards. It is available through the IBM Container Registry (icr.io) and undergoes regular vulnerability scans to ensure security. You can download the image from the [ibm-z-oss-hub/containers repository](#).

Deployment versatility on IBM Z infrastructure refers to the ability to run containerized AI workloads flexibly across various environments within the IBM Z ecosystem. Whether you are deploying in a Linux on IBM Z virtual machine, using z/OS Container Extensions (zCX), or operating within a Red Hat OpenShift cluster, IBM Z supports seamless integration and horizontal scalability.

Because of this flexibility, you can optimize resource utilization, maintain consistent deployment practices, and take advantage of platform-specific features such as high availability, security, and performance. As a result, you can tailor AI inference deployments to meet diverse operational requirements without compromising efficiency or control.

Figure 6-4 on page 70 illustrates three deployment configurations for the Triton Inference Server (Triton IS) on IBM Z infrastructure. Each circular diagram represents a distinct deployment environment:

- ▶ **LinuxONE Configuration:** Shows Triton IS running on LinuxONE, layered between the application and the IBM Z hardware.
- ▶ **zCX Configuration:** Depicts Triton IS deployed within z/OS Container Extensions (zCX), enabling containerized workloads to run alongside traditional IBM z/OS® applications.
- ▶ **Red Hat OpenShift Configuration:** Represents Triton IS operating within a Red Hat OpenShift environment on IBM Z, supporting cloud-native deployment and orchestration.

Together, these configurations highlight the versatility of deploying AI inference workloads across various IBM Z platforms, offering flexibility, scalability, and integration with enterprise systems.

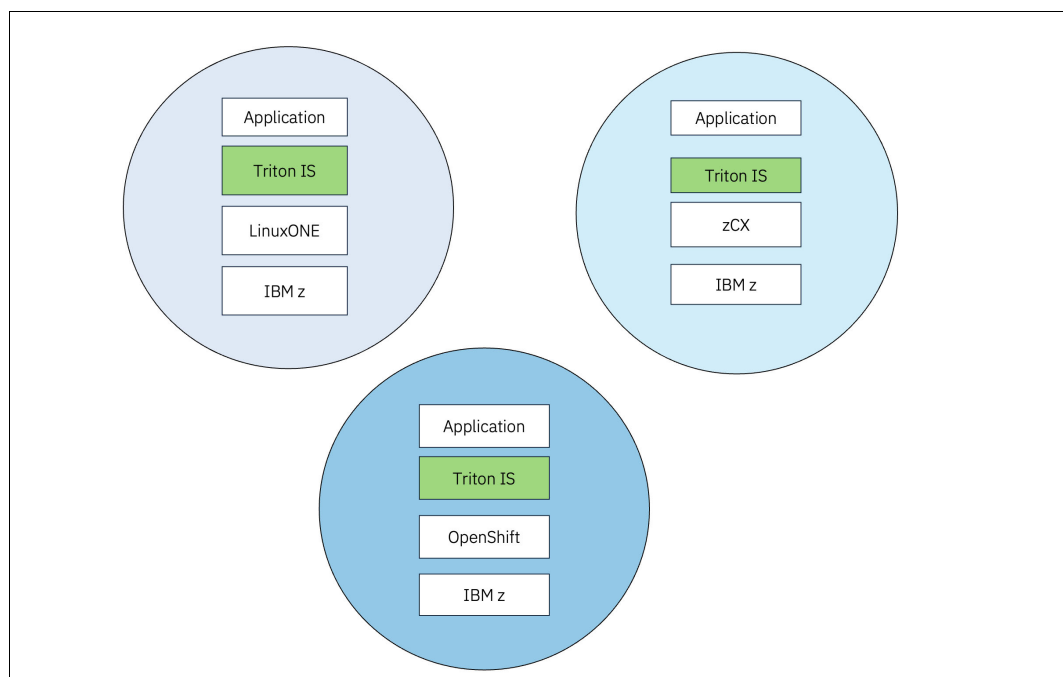


Figure 6-4 IBM Z Accelerated for NVIDIA Triton Inference Server Architecture Deployment strategies in production

6.2.2 Production readiness checklist

Before promoting Triton Inference Server to a production environment, verify that all critical components are properly configured, secured, and tested. Use the following list as a guide to ensure that your deployment meets production-grade standards:

- ▶ Resource limits for CPU and memory are correctly configured for the container.
- ▶ Persistent storage is mapped for the model repository and log files.
- ▶ Health check endpoints are defined and tested.
- ▶ Scaling thresholds and auto-scaling policies are configured.
- ▶ Logging is properly set up and verified.
- ▶ Load testing is completed using expected production workloads.

6.2.3 Model analyzer support for IBM's customized backends

Model Analyzer supports IBM's customized backends for the Triton Inference Server, providing a command-line interface (CLI) tool that helps you optimize configurations for single models, multiple models, ensembles, or models using backend lifecycle scheduling (BLS). This tool identifies the most efficient configuration for your specific IBM Z hardware and generates detailed performance reports. These reports offer insights into the tradeoffs between different configurations, including compute and memory requirements, so you can make informed decisions that maximize performance and resource usage.

For more information and sample usage, see [Model Analyzer for IBM Z](#).

6.3 Sample usage and deployment options

This section describes how to deploy and manage various use cases and model combinations when using Triton Inference Server for enterprise AI workloads.

Figure 6-5 illustrates a structured model repository used with the Triton Inference Server on IBM Z, showcasing multiple backends including ONNX, Snap ML C++, and PyTorch. It highlights how different models such as fraud detection, credit scoring, health insurance analytics, and BERT are organized with their configurations and versioned model files for deployment.



Figure 6-5 Model repository layout for IBM Z–optimized Triton backends.

Example 6-1 on page 72 shows the first portion of the model repository, which includes a fraud-detection model deployed by using the IBM ONNX-MLIR backend. The repository contains three versioned model binary files (`model.so`) and a configuration file (`config.pbtxt`). It also includes an ensemble pipeline directory (`ensemble_fraud_pipeline/`) that contains its own configuration file. This ensemble configuration orchestrates multiple models as part of a composite inference workflow.

The sample model repository demonstrates a traditional ONNX model that was compiled into a `model.so` shared object by using the zDLC compiler and deployed on the TIS-ONNX-MLIR backend with support for model versioning.

Example 6-1 Fraud detection model and ensemble pipeline structure using the IBM ONNXMLIR backend.

```
model_repository/  
+-- on_IBM_ONNXMLIR_Backend/  
|   +-- fraud_detection/  
|       +-- config.pbtxt  
|       +-- 1/  
|           +-- model.so  
|       +-- 2/  
|           +-- model.so  
|       +-- 3/  
|           +-- model.so  
+-- ensemble_fraud_pipeline/  
|   +-- config.pbtxt
```

The next portion of the model repository, the `credit_scoring_model`, is illustrated in Figure 6-5 on page 71 and shown in Example 6-2. This example demonstrates a model in PMML format, with a JSON-based preprocessing pipeline, and is deployed by using the IBM Snap ML C++ backend.

Example 6-2 Credit scoring model deployed using the IBM Snap ML C++ backend with PMML and pipeline configuration

```
+-- from_IBM_Snap_ML_C++_Backend/  
|   +-- credit_scoring_model/  
|       +-- config.pbtxt  
|       +-- 1/  
|           +-- model.pmm1  
|           +-- pipeline.json
```

The sample model repository shown in Example 6-3 demonstrates several PyTorch-based deployments by using IBM's custom backend for Triton Inference Server. The `transaction_risk_classifier` model represents an advanced use case where both model logic and weights are managed explicitly within the backend, enabling fine-grained control over inference behavior. The `bert_base_cased_model` illustrates integration with Hugging Face models using pre-trained checkpoints, which are organized under a structured data directory. Also, the `health_insurance_analytics` model illustrates a custom PyTorch deployment, with separate files for model logic (`model.py`) and weights (`model.pth`). These examples, which are shown in Example 6-3, highlight the flexibility of Triton's PyTorch backend in supporting diverse enterprise AI workloads on IBM Z.

Example 6-3 PyTorch models deployed on IBM Triton with custom logic and Hugging Face integration.

```
+-- from_IBM_PyTorch_Backend/  
|   +-- health_insurance_analytics/  
|       +-- config.pbtxt  
|       +-- 1/  
|           +-- model.py  
|           +-- model.pth  
|   +-- bert_base_cased_model/  
|       +-- config.pbtxt  
|       +-- data/  
|           +-- bert-base-cased/  
|               +-- config.json  
|               +-- pytorch_model.bin
```

```
+-- tokenizer_config.json
+-- special_tokens_map.json
+-- vocab.txt
+-- tokenizer.json
```

6.4 End-to-end model deployment use cases with IBM TIS

To support scalable, real-time AI workloads, the IBM Triton Inference Server (TIS) can be deployed across multiple IBM-customized backends, each optimized for specific model types and performance objectives. This section describes how a fraud-detection use case can be implemented by using end-to-end deployment pipelines across the IBM Snap ML C++, PyTorch, and ONNX-MLIR backends.

Although this section does not include complete code listings, it references publicly available repositories that demonstrate how to configure and deploy these models effectively by using Triton on IBM Z infrastructure.

TDeployment on IBM Triton Inference Server Snap ML C++ Backend

This example demonstrates an end-to-end deployment and inference pipeline using the IBM Triton Inference Server Snap ML C++ Backend. It includes both data preprocessing and model prediction, showcasing how to use the high performance of Snap ML C++ runtime for low-latency inference.

See the following repository: [ibm-snapml-backend-example](#)

Deployment on IBM Triton Inference Server PyTorch Backend

This example illustrates the complete deployment workflow using the IBM Triton Inference Server PyTorch Backend. It covers model serving and inference for PyTorch models, enabling integration with Triton Inference Server for dynamic and flexible model execution.

See the following repository: [ibm-pytorch-backend-example](#)

Deployment on IBM Triton Inference Server-ONNX MLIR Backend

This example presents an end-to-end deployment pipeline for ONNX models using the IBM Triton Inference Server ONNX-MLIR Backend. It demonstrates how to compile and serve ONNX models efficiently, using the MLIR-based backend for optimized inference performance.

See the following repository: [ibm-onnxmlir-backend-example](#)

6.5 IBM Z accelerated for TIS: Best practices

This section outlines operational strategies and configuration recommendations for optimizing the deployment and performance of models on the TIS. It provides a practical guide for machine learning engineers and DevOps teams to enhance inference efficiency, reduce latency, and improve observability in production environments.

6.5.1 Defining response cache in TIS

The response cache feature in TIS enables caching of inference responses, so identical future requests can be served directly from the cache without re-running the model. This significantly reduces latency and improves throughput, especially in workloads with repetitive queries.

Response caching can be configured at both the model level and the server level, providing flexibility in deployment. Configuration options are illustrated in Example 6-4.

Example 6-4 Response cache configuration

```
# Server-level caching
tritonserver --cache-config local, size=1048576

# Model-level caching in config.pbtxt
response_cache {
    enable: true
}
```

For detailed setup instructions and advanced options, see [Triton Response Cache](#).

6.5.2 Using the model analyzer for performance optimization

The Model Analyzer tool in the Triton Inference Server is designed to evaluate and optimize model performance across different deployment configurations. It systematically runs models with various settings, such as batch size, concurrency levels, and instance counts, and collects key performance metrics including throughput, latency, and memory utilization.

By analyzing these metrics, you can identify the most efficient configuration for your models, ensuring optimal resource usage and responsiveness in production environments. This tool is especially valuable for fine-tuning AI workloads on IBM Z to achieve balanced performance and scalability.

6.5.3 Enabling dynamic batching in the configuration file

Dynamic batching is a performance optimization feature in Triton Inference Server that groups individual inference requests into batches dynamically at runtime. This improves throughput and processor usage by maximizing the efficiency of model execution, especially under high-concurrency workloads.

Dynamic batching is configured in the model's config.pbtxt file. The `max_queue_delay_microseconds` parameter defines the maximum time Triton waits to form a batch. For example, setting this value to 100000 (100 ms) allows the server to aggregate incoming requests within that time window before dispatching them as a batch. Configuration options are illustrated in Example 6-5.

Example 6-5 Dynamic batching configuration

```
dynamic_batching {
    max_queue_delay_microseconds: 100
}
```

Model warmup is a feature in Triton Inference Server that helps eliminate latency spikes associated with the first inference request after model loading. By simulating inference

requests during server startup, model warmup ensures that models are fully initialized and ready to serve traffic with optimal performance from the outset.

Model warmup is configured in the model's `config.pbtxt` file. Example 6-6 demonstrates a warmup configuration named `production_warmup` that uses a batch size of 8 and generates random input data for the tensor named `input_tensor`.

Example 6-6 Model warmup configuration

```
model_warmup [  
  {  
    name: "production_warmup"  
    batch_size: 8  
    inputs {  
      key: "input_tensor"  
      value {  
        data_type: TYPE_FP32  
        dims: [224, 224, 3]  
        random_data {}  
      }  
    }  
  }  
]
```

6.5.4 Using the HTTP & gRPC end points exposed

Triton Inference Server exposes both HTTP and gRPC endpoints, which are essential for managing models and monitoring server health in production environments. These standardized interfaces can help with integration with client applications and orchestration tools.

These endpoints support a variety of operational tasks:

- ▶ Server health checks to verify readiness and liveness
- ▶ Model metadata retrieval and status inspection
- ▶ Dynamic model loading and unloading at run time

By using these endpoints, users can automate model lifecycle management and ensure high availability of inference services. For detailed API specifications and usage examples, see [Triton HTTP/gRPC API](#).

6.5.5 Running multiple model instances

To fully use available compute resources, Triton Inference Server supports running multiple instances of a model using the `instance_group` configuration. This enables parallel execution of inference requests, improving throughput and reducing latency under high-load conditions.

Example 6-7 provides a configuration example that defines two CPU-based model instances.

Example 6-7 Multiple model instances

```
instance_group [  
  {  
    count: 2  
    kind: KIND_CPU  
  }  
]
```

6.5.6 Using logging options for better monitoring and debugging

Effective logging is essential for monitoring, debugging, and maintaining production-grade inference services. Triton Inference Server provides configurable logging options that allow users to control the verbosity and type of messages generated at run time.

The following logging flags can be set to tailor the output:

- ▶ **--log-verbose:** Controls debug-level logging. Values range from 0 (disabled) to 2 (maximum verbosity).
- ▶ **--log-info:** Enables general informational messages useful for tracking server activity.
- ▶ **--log-warning:** Displays warnings that do not interrupt model serving but might indicate potential issues.
- ▶ **--log-error:** Logs critical errors that require attention.

These options help operators gain visibility into server behavior, diagnose performance bottlenecks, and ensure reliable operation. For detailed configuration and usage, see [Triton Logging Extension](#).



IBM Z Accelerated for TensorFlow

As the number of customers, transactions, transaction types, and vendors continues to grow, detecting fraudulent activity in near real time becomes increasingly complex. To address these challenges, many organizations use deep neural networks implemented through open-source frameworks such as TensorFlow. These networks are suited for identifying complex patterns in large datasets and often produce more accurate inferences with fewer false positives than traditional decision-based systems. You can continually update the model to adapt to new forms of fraud as they emerge.

By using IBM Z Accelerated for TensorFlow includes TensorFlow and the IBM-zDNN-Plugin developed specifically for TensorFlow on IBM Z. This chapter introduces TensorFlow, explains how to create, train, save, and load models, and examines the optimizations available for inference through the IBM-zDNN-Plugin. It also presents a use case that creates and trains a model to perform near real-time credit-card fraud detection by using TensorFlow, followed by optimized inference calls that use the IBM-zDNN-Plugin.

This chapter includes the following topics:

- ▶ 7.1, “Introduction to TensorFlow” on page 78
- ▶ 7.2, “Model training and saving” on page 80
- ▶ 7.3, “IBM-zDNN-Plugin” on page 83
- ▶ 7.4, “Near real-time credit card fraud detection use case” on page 85

7.1 Introduction to TensorFlow

TensorFlow is an open-source machine-learning framework that is widely used in the industry to create, train, and run inference on neural-network models. The Google Brain team, the artificial-intelligence research group at Google, developed TensorFlow and released it as open source in 2015.

TensorFlow provides a predefined set of operations through a Python interface. Each operation uses one or more C++-implemented kernels for efficient computation. By using this design, you can register kernels that handle device-specific and data-type-specific computations for each operation while offering a simple Python interface.

TensorFlow primarily focuses on operations used in deep neural networks. It uses the Eigen C++ library for many CPU kernels and CUDA for GPU kernels.

7.1.1 Models

TensorFlow primarily uses Keras, an open-source Python library, to create models. These models and their underlying layers consist of multiple TensorFlow operations. By using this design, you can create models by using predefined Keras layers or develop custom layers by using TensorFlow operations to fit specific use cases.

You can train these models, use them immediately for inference, or save them to a file for later loading to continue training or perform inference.

TensorFlow also has a strong community that provides pretrained models you can use without modification to perform many common machine-learning tasks. Keras Applications, TensorFlow Hub, and Hugging Face are widely used model repositories. You can use the repositories to develop and test machine-learning use cases quickly by using only a few lines of Python code.

7.1.2 Graph execution

TensorFlow was originally designed around computational graphs, where operations are represented as nodes and data flows between them as tensors. This design helps make it easier to serialize models for saving and you can run them outside of Python environments. Figure 7-1 on page 79 provides a visual representation of a TensorFlow graph.

When you use graph execution mode, TensorFlow performs several optimizations on the entire graph during the first execution to create an optimized path for all subsequent runs. This can be especially beneficial when you use the model with multiple inputs that are grouped into mini-batches.

TensorFlow includes the following optimizations:

- ▶ Constant Folding: Nodes with inputs that are static values can be replaced with computed values.
- ▶ Parallelism: Independent subgraphs within a graph can be split into parallel threads or devices.
- ▶ Remapping: Common subgraph expression can be remapped into a single fused node.

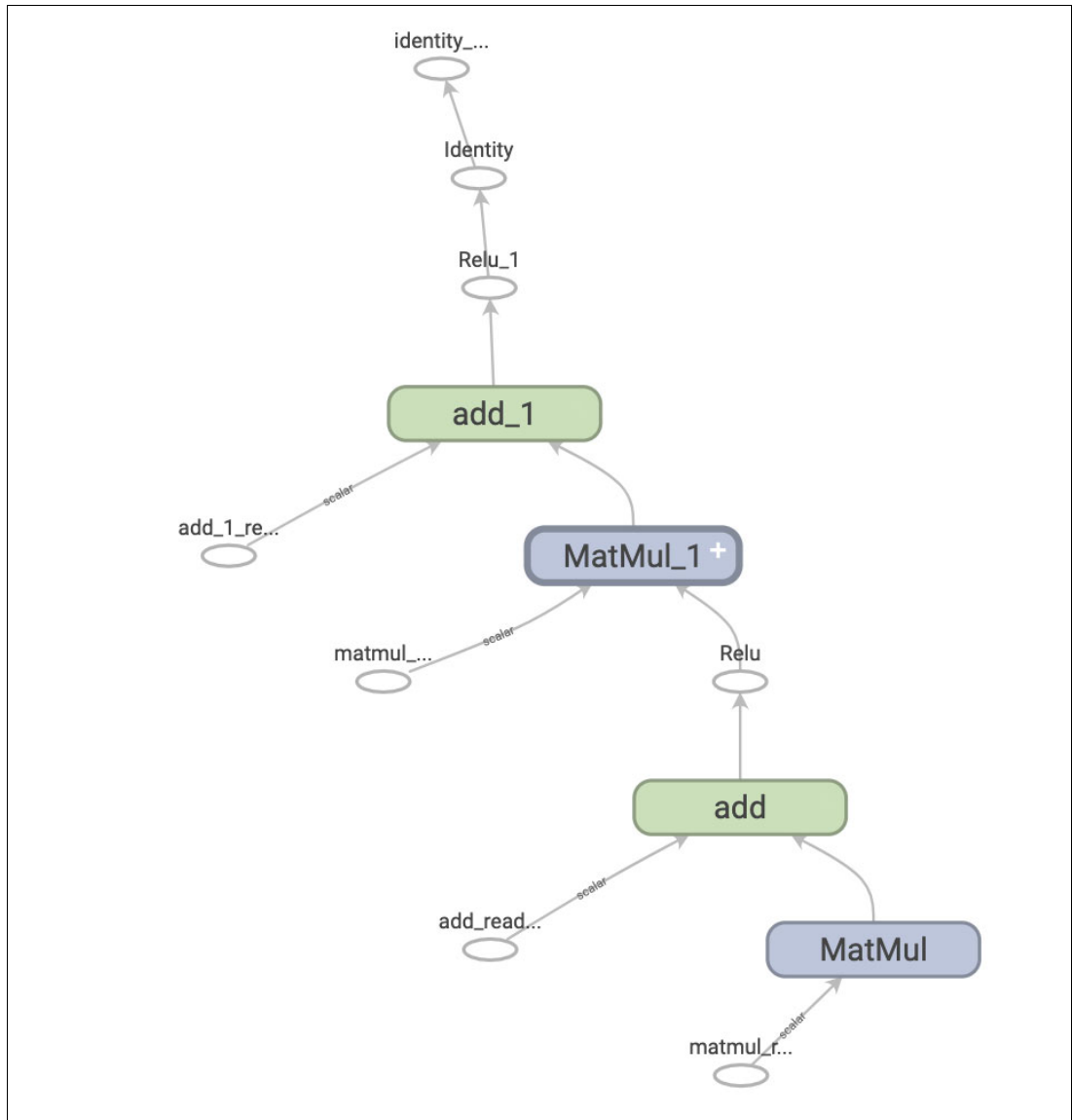


Figure 7-1 Visual representation of a TensorFlow Graph

7.1.3 Eager execution

TensorFlow version 2.0 and later also offers eager execution mode, which performs operations one at a time without modification. With this mode, you can quickly run models during development and debugging without needing to modify them.

When your model is ready for production, you can switch to graph execution mode to take advantage of optimized execution paths.

7.1.4 TensorFlow functions

By default, Keras models use graph execution within the built-in functions for model training and inferencing. Also, when you use TensorFlow, you can run any model or Python function in graph execution mode by using a TensorFlow Function. Example 7-1 on page 80 demonstrates how to convert a regular Python function into a TensorFlow graph function.

Example 7-1 Converting a Python function to a TensorFlow graph function

```
import tensorflow as tf
def matmul_add(a, b, c):
    return tf.matmul(a, b) + c
graph_func = tf.function(matmul_add)
result = graph_func(a, b, c)
```

For convenience, TensorFlow provides a decorator, so you can mark any function to run in graph execution mode, as shown in Example 7-2.

Example 7-2 Using the @tf.function decorator to enable graph execution

```
@tf.function
def matmul_add(a, b, c):
    return tf.matmul(a, b) + c
result = matmul_add(a, b, c)
```

7.2 Model training and saving

Creating a model, training it, and saving its structure and weights involve multiple steps. However, Keras provides built-in methods so that you can handle these tasks more easily. This section provides an overview of model creation, training, saving, and exporting.

7.2.1 Model creation

A Keras model consists of one or more layers. The first layer serves as the input layer, and the final layer serves as the output layer. You pass these layers to the Model constructor when you create the model. Example 7-3 defines a simple feed-forward neural network using Keras' functional API in TensorFlow.

Example 7-3 Defining a binary classification model using Keras functional API

```
import tensorflow as tf
import tensorflow.keras as keras
inputs = keras.Input(shape=[220,])
x = keras.layers.Dense(units=200, activation="sigmoid")(inputs)
x = keras.layers.Dense(units=200, activation="tanh")(x)
outputs = keras.layers.Dense(units=1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

7.2.2 Model training

After you construct the model, you train it on a dataset by first calling the compile method with an optimizer, loss function, and metrics. Then, you call the fit method with the training data. Keras provides a wide variety of built-in optimizers, loss functions, and metrics so that you can train many models with just a few lines of code.

Example 7-4 shows how to compile and train a Keras model.

Example 7-4 Compiling and training a Keras model

```
model.compile(optimizer="adam",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(train_dataset, epochs=20)
```

7.2.3 Model saving and exporting

After you train the model, you can serialize and save it to a file for later use in training or inference. Keras provides a built-in method to save the model and its current state, as shown in Example 7-5.

Example 7-5 Saving a Keras model in the native .keras format

```
model_path = "./saved_model/dense.keras"
model.save(model_path)
```

When you no longer need to train the model, you can export it exclusively for inference. The exported TensorFlow SavedModel cannot be used for further training. Keras provides a built-in method to export the model, which creates a SavedModel at the specified path. This format includes all the files that are needed to serve the model in a production environment such as TensorFlow Serving, which is discussed further in Chapter 8, “IBM Z Accelerated Serving for TensorFlow” on page 89.

Example 7-6 shows how to export a trained Keras model in a format suitable for TensorFlow Serving, which is used to deploy models in production environments.

Example 7-6 Exporting a model for TensorFlow Serving

```
version = 1
servable_path = f"/serving_model/dense/{version}/"
model.export(servable_path)
```

7.2.4 Model loading and inference

You can load a saved Keras model to continue training or perform inference. Keras provides a built-in function to load a saved model:

```
model = keras.models.load_model(model_path)
```

The returned model can be used for inference in two ways:

1. Eager execution mode: Call the model directly with a batch of inputs:

```
pred = model(test_batch) # eager execution mode
```

2. Graph execution mode: Use the built-in predict method to run inference on multiple batches:

```
pred = model.predict(test_dataset) # graph execution mode
```

Although the TensorFlow SavedModel format is used primarily with TensorFlow Serving, you can also load it directly in TensorFlow for inference or evaluation. Example 7-7 on page 82 demonstrates how to load a TensorFlow SavedModel and use it for inference.

Example 7-7 Loading and serving a TensorFlow SavedModel

```
model = tf.saved_model.load(model_path)
pred = model.serve(test_batch)
```

7.2.5 Model profiling

To understand and optimize the performance of TensorFlow models, especially in production environments, you must analyze how computational resources are used during execution. Profiling provides valuable insight into a model's runtime behavior. It helps you identify bottlenecks, optimize execution paths, and validate the impact of hardware accelerators such as the IBM-zDNN-Plugin (IBM Z Deep Neural Network Library), which is discussed in section 7.3, "IBM-zDNN-Plugin" on page 83.

TensorFlow includes built-in tools that guide you through the profiling process and enable you to capture detailed performance metrics during model inference and training.

Capturing profiling information

You can capture profiling information by enabling the TensorFlow built-in profiler before and after the specific section of code you want to analyze. Example 7-8 shows capturing profiling data in TensorFlow by starting and stopping the profiler around a model inference operation.

Example 7-8 Capturing profiling data

```
tf.profiler.experimental.start(logdir=path_to_save_to)
pred = model.predict(test_dataset)
tf.profiler.experimental.stop()
```

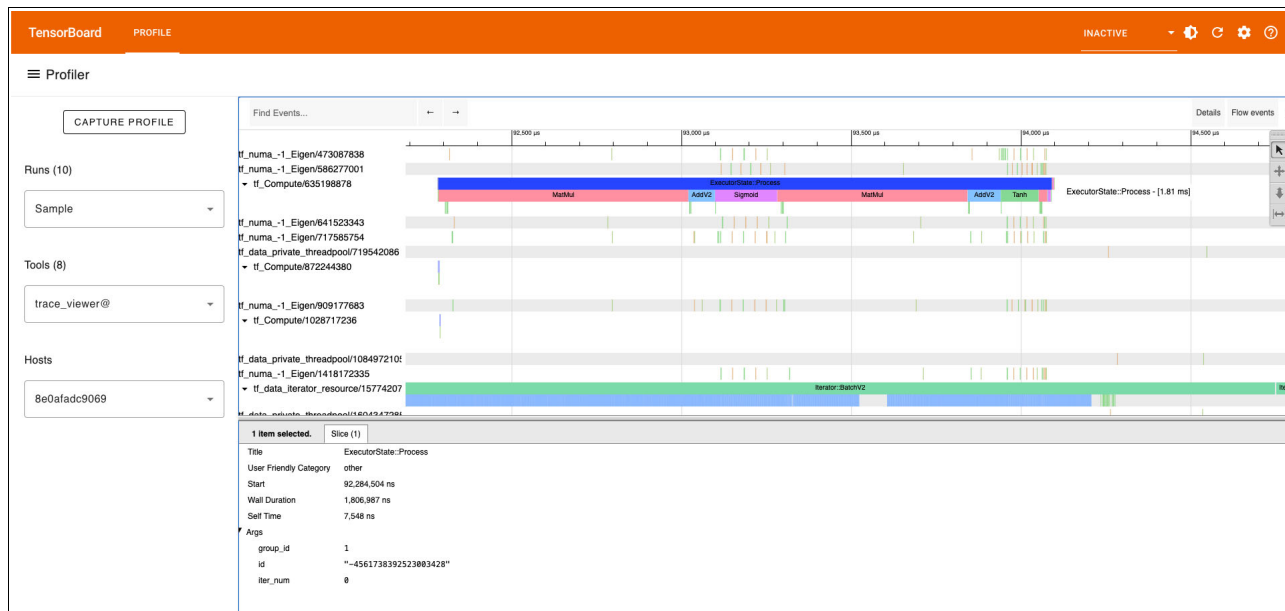
This command creates a directory at the specified path, where TensorFlow stores the captured profiling data.

Viewing profiling information

You can view the captured profiling data by using TensorBoard, an open-source Python tool. To start it from the console, run the following command and specify the path to the profiling information captured in Example 7-8:

```
tensorboard --logdir <path to captured profiling information>
```

After you initialize the profiler, you can view the profiling data in a web browser by navigating to <http://localhost:6006/>. From there, open the `trace_viewer` tool to explore a visual representation of the profiling data, including detailed metrics for each operation, as shown in Figure 7-2 on page 83.



In this view, you can examine the underlying operations of each model layer, including matrix multiplication, addition, and activation functions across all three Dense layers.

7.3 IBM-zDNN-Plugin

Although TensorFlow provides a predefined framework for building neural networks, you can register additional operations, kernels, and graph optimizations when TensorFlow initializes. The IBM-zDNN-Plugin automatically registers custom operations that are commonly used in neural-network models, along with kernel implementations that are optimized for the IBM Telum processor.

The plugin also applies graph-level optimizations that replace standard TensorFlow operations with these custom implementations. When you use this process, you do not need to modify existing models or code to take advantage of the hardware acceleration provided by IBM Z.

7.3.1 Integration of IBM-zDNN-Plugin into IBM Z

The IBM Telum processor, introduced with IBM z16, represents a major advancement in enterprise microprocessor technology. It integrates deep-learning inference capabilities directly into the hardware, enabling real-time fraud detection and faster decision-making.

You can access these hardware features through the IBM Z Deep Neural Network (zDNN) library, which provides a C interface to the hardware instructions and includes functions for preparing data to meet the requirements of the Telum processor.

The IBM-zDNN-Plugin registers custom operations in TensorFlow that use zDNN to preprocess input data, run computations on the Telum Processor, and postprocess the output. By using this integration, you can accelerate AI workloads without changing existing models or code.

Custom operations

Custom operations extend the predefined set of operations of TensorFlow to align with the hardware instructions available on the IBM Telum processor. These operations are essential to the integration of the IBM-zDNN-Plugin, enabling you to offload supported computations to the processor's on-chip AI accelerator.

Because the Telum processor has specific data-format requirements, the plug-in also registers noncomputational operations to manage data preprocessing and postprocessing. This approach prevents redundant transformations and helps ensure efficient execution.

Each custom operation mirrors a standard TensorFlow operation but uses a name that is prefixed with “Z” (for example, ZMatMul). This naming convention enables TensorFlow to substitute standard operations with hardware-optimized equivalents during graph execution without requiring changes to your model code.

Kernel implementations

Each custom operation includes one or more kernel implementations that handle device-specific and data-type-specific computations. These kernels currently support both 32-bit and 16-bit floating-point data types.

When you run a model, each kernel validates the input and output tensors. If the operation is supported by zDNN, the kernel uses the AI acceleration of the IBM Telum processor. If the operation is not supported, the kernel automatically falls back to the default Eigen implementation provided by TensorFlow. This approach ensures compatibility, and you can benefit from hardware acceleration when available.

Graph optimizations

The IBM-zDNN-Plugin includes a custom graph optimizer that works alongside the built-in optimization framework of TensorFlow. When you run a model, this optimizer automatically replaces eligible TensorFlow operations with custom operations that meet the Telum processor's data requirements.

This substitution enables the model to use hardware acceleration during run time without requiring changes to your code.

Environment variables

When you install the IBM-zDNN-Plugin, the registered graph optimizer automatically replaces supported TensorFlow operations with custom operations that use the Telum processor's AI acceleration.

In some cases, such as during model training, you might want to disable these graph optimizations. You can disable them by setting an environment variable before initializing TensorFlow:

```
export NNPA_DEVICES=0
```

For more information about the IBM-zDNN-Plugin and its integration with IBM Z Accelerated for TensorFlow, see <https://github.com/IBM/ibmz-accelerated-for-tensorflow>.

7.3.2 Model profiling

Running the same sample code that is shown in Example 7-8 on page 82 with the IBM-zDNN-Plugin installed produces a similar view of the captured profiling information, which is shown in Figure 7-3 on page 85.

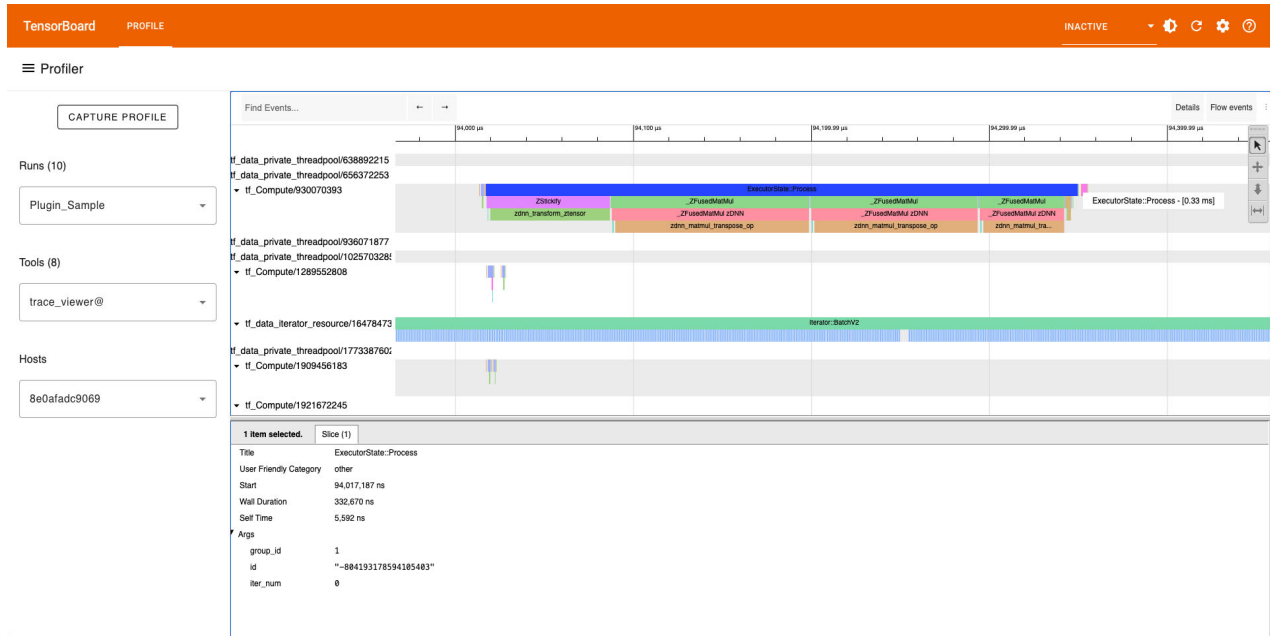


Figure 7-3 Sample trace view with IBM-zDNN-Plugin using TensorBoard

When you examine the profiling data, you can see that the matrix-multiplication, addition, and activation functions from each of the three Dense layers are fused into a single custom operation named `_ZFusedMatMul`. The kernel implementation for this operation validates the input and output tensors and, when supported, starts the `zdnm_matmul_transpose_op` function to perform the computation by using the IBM zDNN library.

In addition to the fused operations, the graph includes two custom transformation operations: `ZStickify` and `ZUnstickify`. These operations convert data to and from the format required by the Telum processor's AI accelerator. `ZStickify` prepares the input data before the first Dense layer, and `ZUnstickify` restores the output format after the third layer.

Although these data transformations introduce some additional processing, the overall runtime in this example is significantly reduced from 1.80 milliseconds to 0.33 milliseconds, which demonstrates the performance benefits of hardware acceleration with zDNN.

7.4 Near real-time credit card fraud detection use case

Credit-card fraud detection is a critical application of AI in the financial sector, where rapid and accurate decision-making can prevent significant financial losses. By using deep-learning models on IBM Z, you can analyze transaction patterns in real time while maintaining data locality, security, and compliance.

This section presents a use case that illustrates near real-time credit-card fraud detection by predicting whether a transaction is fraudulent based on the current transaction and the six preceding transactions. Because the input data is temporal, this example uses a Recurrent Neural Network (RNN) to model sequential patterns in transaction behavior.

You can access the sample code for this use case at:

<https://github.com/IBM/ibmz-accelerated-for-tensorflow/tree/main/samples/credit-card-fraud>.

7.4.1 Background information

As discussed in section 2.3, “Credit Card Fraud use case overview” on page 13, the dataset for this use case is provided as a comma-separated values (CSV) file that contains credit-card transaction information.

The transactions are sorted by user and date-time, allowing consecutive user transactions to be accessed efficiently. These sorted transactions are then split into training, validation, and testing sets.

Because the dataset contains far more nonfraudulent transactions than fraudulent ones, the training set uses weighted sampling when producing batches to ensure that the model is trained with an equal distribution of both types of transactions.

7.4.2 Model construction and training

The model for this use case is implemented by using Keras, which provides built-in support for both Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) layers. These are two common types of Recurrent Neural Networks (RNNs) that are suited for processing sequential data. In this example, LSTM layers are used, although GRU layers can be substituted with minimal changes because of their similar functionality.

The architecture consists of two stacked LSTM layers, where the output of the first layer is passed directly to the second. The final output is then fed into a Dense layer with a sigmoid activation function to perform binary classification (see Example 7-9).

Example 7-9 Using LSTM layers

```
model = keras.Sequential([
    keras.layers.Input(shape=(220,)),
    keras.layers.LSTM(units=200, return_sequences=True),
    keras.layers.LSTM(units=200, return_sequences=False),
    keras.layers.Dense(units=1, activation="sigmoid")
])
```

The model is trained for 20 epochs using 50,000 balanced batches per epoch, ensuring equal representation of fraudulent and nonfraudulent transactions. Training can be run by using the [credit_card_fraud_training.py](#) script that is provided in the sample repository.

7.4.3 Model profiling

You can perform inference by using the [credit_card_fraud.py](#) script. To establish a performance baseline, the model is first run with the IBM-zDNN-Plugin graph optimizer disabled.

When you review the profiling information for a single batch (Figure 7-4 on page 87), you can see the underlying operations for the two LSTM layers and the final Dense layer. The built-in graph optimizer recognizes that the leading three matrix multiplications of the LSTM layer are independent and splits them into separate threads that run in parallel.

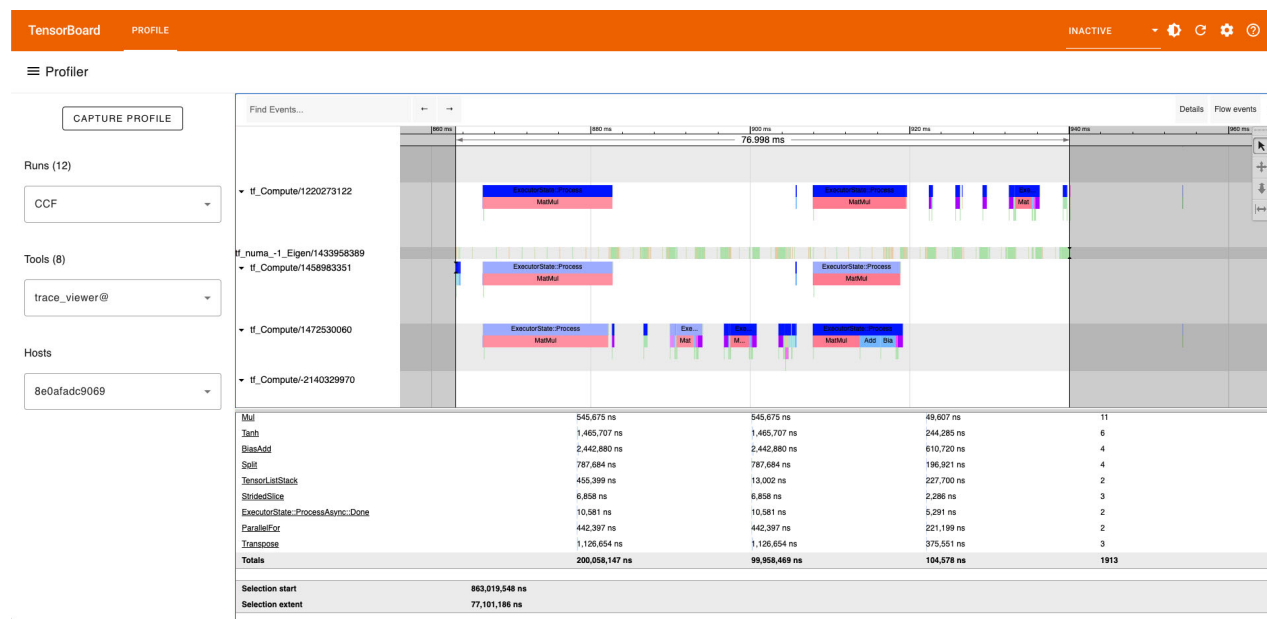


Figure 7-4 LSTM trace view with NNPA_DEVICES=0 using TensorBoard

Running the use case code with the IBM-zDNN-Plugin graph optimizer enabled produces a similar view of the captured profiling information, shown in Figure 7-5.

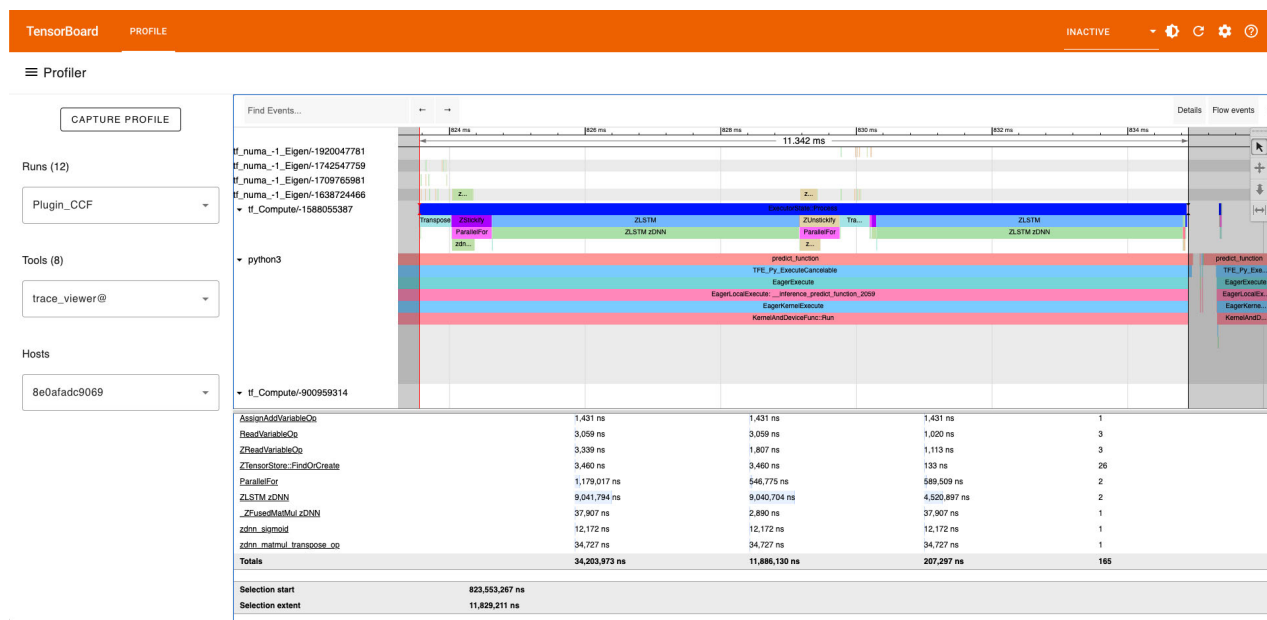


Figure 7-5 LSTM trace view with NNPA_DEVICES=1 using TensorBoard

The IBM-zDNN-Plugin graph optimizer fuses each of the LSTM layers into a single custom operation, ZLSTM. With these fused operations that use zDNN, the runtime for this entire batch is reduced from 77.00 ms to 11.34 ms.



IBM Z Accelerated Serving for TensorFlow

Deploying machine-learning models in production environments requires more than training accurate models. It requires a scalable, secure, and efficient serving infrastructure. Although TensorFlow provides a comprehensive framework for model development, real-world deployment introduces challenges such as managing frequent updates, scaling inference workloads, and integrating with client-facing APIs.

To address these requirements, TensorFlow Serving was developed to streamline the deployment and serving of TensorFlow models. IBM Z Accelerated Serving for TensorFlow builds on this foundation and enables clients to create production-ready environments that take full advantage of IBM Z performance, security, and reliability.

This chapter introduces TensorFlow Serving and explains how the IBM-zDNN-Plugin enhances inference performance. A practical use case for near real-time credit-card fraud detection illustrates how these technologies work together to deliver optimized AI solutions on IBM Z.

This chapter includes the following topics:

- ▶ 8.1, “Introduction to TensorFlow Serving” on page 90
- ▶ 8.2, “Near real-time credit card fraud detection use case” on page 93

8.1 Introduction to TensorFlow Serving

TensorFlow Serving is an open-source system designed to support the deployment and inference of neural-network models in production environments. It extends the runtime capabilities of TensorFlow by providing advanced features such as model versioning for seamless updates, dynamic batching to optimize inference throughput, and robust model-management APIs that enable serving multiple models concurrently.

In addition, TensorFlow Serving is framework-independent so that models developed outside of TensorFlow can be served with minimal integration effort. Figure 8-1 illustrates the architecture and workflow of TensorFlow Serving, showing how models (servables) are managed and served in a production environment.

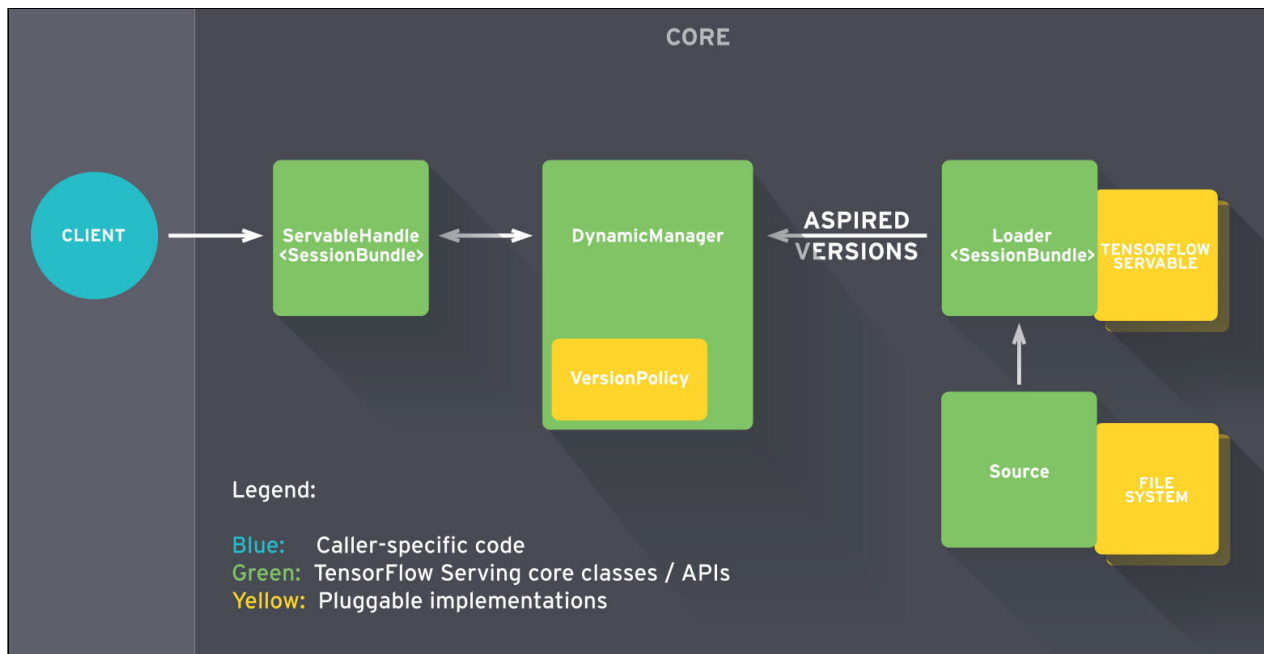


Figure 8-1 TensorFlow Serving architecture

8.1.1 Servable

A servable is a core abstraction in TensorFlow Serving that represents any object made available to clients for performing a specific task, such as a machine-learning model or a lookup table. One of the most common types of servables is the TensorFlow SavedModel, which is the default format for saving TensorFlow models and for exporting Keras models.

Each servable is versioned, so multiple versions can coexist. By default, TensorFlow Serving loads and serves the latest available version from the file system, enabling seamless model updates without service interruption.

8.1.2 ModelServer

ModelServer is the core component in TensorFlow Serving that is responsible for loading servables from the file system, managing multiple versions of each servable, and handling inference requests received through client APIs. It is typically initialized by using the command-line interface, where the only required arguments are the model name and the path to the directory that contains the servable versions.

In addition to these required parameters, the ModelServer supports a variety of optional arguments so that you can customize its behavior. The following options are commonly used:

- ▶ **--port**: Specifies the port to listen on for the gRPC API.
- ▶ **--rest_api_port**: Specifies the port to listen on for the HTTP/REST API.
- ▶ **--model_config_file**: Specifies the path to a ModelServerConfig file that can be used to define multiple models, each with its own configuration.
- ▶ **--enable_batching**: Enables automatic batching of multiple requests into a single request. This option requires that a batching parameters file be provided.
- ▶ **--batching_parameters_file**: Specifies the path to a BatchingParameters file that defines the constraints under which multiple requests are batched into a single request.

After the latest version of a servable is loaded, TensorFlow Serving continues to monitor the model directory for new versions. When a new version becomes available, it is automatically loaded and served as the latest version. Alternatively, specific versions (single or multiple) can be explicitly defined by using a ModelServerConfig file.

By default, servables are loaded lazily, meaning they are initialized only when the first request is received. As a result, initial requests might experience higher latency because of model loading and graph optimization. To mitigate this effect, TensorFlow Serving supports SavedModel warmup files. When present, these files trigger a series of warmup runs during server startup, reducing the latency typically observed during the first inference requests.

For more information about configuring a ModelServer, see [Tensorflow Serving Configuration](#).

8.1.3 Client APIs

TensorFlow Serving provides APIs that enable clients to retrieve model metadata and perform inference requests. Both REST and gRPC interfaces are supported, offering flexibility depending on the use case. The REST API offers ease of use and accessibility, whereas gRPC provides higher performance and efficiency, particularly in high-throughput environments.

Each interface exposes a consistent set of endpoints. The endpoints are as follows:

- ▶ **Model Status**: Returns the status of the model in the ModelServer
- ▶ **Model Metadata**: Returns metadata, including the inputs and outputs of the model in the ModelServer
- ▶ **Classify/Regress**: Performs inference on the model in the ModelServer by using a single input and returns the classification and score
- ▶ **Regress**: Performs inference on the model in the ModelServer by using a single input and returns the regression result
- ▶ **Predict**: Performs inference on the model in the ModelServer by using inputs and outputs with signatures defined in the model metadata

8.1.4 Runtime

TensorFlow Serving uses TensorFlow for the computation of models, providing the same operations, computational kernels, and graph optimization available when using the Python interface. For IBM Z Accelerated Serving for TensorFlow, the custom operations, computational kernels, and graph optimizations from IBM-zDNN-Plugin are also included within TensorFlow Serving.

8.1.5 Model deployment

Deploying a model with TensorFlow Serving involves creating a Servable. As discussed previously, the default format used by TensorFlow's built-in saving methods, SavedModel, is already a valid Servable. This means that any model saved by using TensorFlow's standard APIs can be deployed directly with TensorFlow Serving without modification.

As noted in Chapter 7, "IBM Z Accelerated for TensorFlow" on page 77, TensorFlow primarily uses Keras for model creation and serialization. In addition to the `save()` method, Keras models also provide an `export()` method, which exports the model in the SavedModel format required by TensorFlow Serving.

Exporting a Keras model to a TensorFlow SavedModel Servable for deployment with TensorFlow Serving is shown in Example 8-1.

Example 8-1 Exporting a Keras model as a Servable

```
version = 1
servable_path = f"/serving_model/sample_model/{version}/"
model.export(servable_path)
```

8.1.6 Model serving

When a model is deployed as a Servable, you can serve it by using the command-line interface. Example 8-2 illustrates how to start a TensorFlow ModelServer by using the command-line interface to serve a machine learning model.

Example 8-2 Starting a TensorFlow ModelServer instance

```
tensorflow_model_server \
  --port=8500 \
  --rest_api_port=8501 \
  --model_name=${MODEL_NAME} \
  --model_base_path=${MODEL_BASE_PATH}/${MODEL_NAME}
```

This command loads the latest version of the Servable and makes it accessible to clients through gRPC on port 8500 and REST on port 8501.

For IBM Z Accelerated Serving for TensorFlow, this command runs automatically when the container starts. The `MODEL_BASE_PATH` and `MODEL_NAME` values are passed as environment variables to allow flexible configuration. Additional arguments that you provide at container startup are forwarded to the TensorFlow ModelServer command-line interface to enable further customization of the serving environment.

8.1.7 Model Inferencing

When a model is successfully deployed, you can verify its status by using the TensorFlow Serving REST API:

```
curl http://localhost:8501/v1/models/sample_model
```

This command queries the model server to confirm that the model is loaded and available for inference.

You can make inference requests by sending input data as a JSON-encoded string to the server's REST endpoint. Example 8-3 demonstrates how to use a REST API request to perform inference with a model served by TensorFlow Serving.

Example 8-3 Performing inference using the REST API

```
curl -X POST \
  -d '{"instances": [2.0, 5.0, 3.0]}' \
  http://localhost:8501/v1/models/sample_model:predict
```

Inference by using the gRPC API is slightly more complex than using REST, and it is typically performed in Python. To simplify this process, the TensorFlow Serving API Python package is available. It provides tools for many common serving tasks, including constructing and sending inference requests over gRPC.

In addition to the tools that the TensorFlow Serving API Python package provides, metadata describing the model's inputs and outputs is required to construct a valid PredictRequest. You can retrieve this metadata by using the REST API:

```
curl http://localhost:8501/v1/models/sample_model/metadata
```

This request returns information about all available signatures of the model, including the names and data types of the inputs and outputs. This metadata is essential for correctly populating the PredictRequest, which you then pass to a PredictionServiceStub to perform inference through gRPC.

8.2 Near real-time credit card fraud detection use case

This use case demonstrates how to deploy a trained credit card fraud detection model by using IBM Z Accelerated Serving for TensorFlow. The model, originally described in Chapter 7, “IBM Z Accelerated for TensorFlow” on page 77, is served using the IBM Z optimized serving infrastructure and accessed via the gRPC API for high-performance inference.

To view the code that is used in this deployment scenario, see the official IBM sample [ibmz-accelerated-serving-for-tensorflow/samples/credit-card-fraud/](https://github.com/ibmz-accelerated-serving-for-tensorflow/samples/credit-card-fraud/).

8.2.1 Model deployment

The model described in Chapter 7, “IBM Z Accelerated for TensorFlow” on page 77 was saved as a Keras model. Before you serve the model, export it to a SavedModel. You can perform this export by running the script `credit_card_fraud_deployment.py` (See [credit_card_fraud_deployment.py](#)). After you run the script, a new folder is created that contains the serving model in SavedModel format.

The script also adds a SavedModel Warmup file to the SavedModel. This file is a PredictionLogs file that includes a set of sample inputs. These inputs run when the model initializes. By including a SavedModel Warmup file with representative sample inputs, you can reduce the latency that typically occurs during the first inference requests.

8.2.2 Model serving

You can serve the model by launching the IBM Z Accelerated Serving for TensorFlow container. To do this, map ports 8500 and 8501, mount the serving model directory, and set the model name as an environment variable.

Example 8-4 illustrates how to start the IBM Z Accelerated Serving for TensorFlow container by using Docker. It maps the required ports, mounts the model directory, and sets the model name by using an environment variable.

Example 8-4 Launching the serving container with model directory and ports mapped

```
docker run -it --rm --detach \  
  -p 8500:8500 \  
  -p 8501:8501 \  
  -v 'credit-card-fraud/serving_model:/models:z' \  
  -e MODEL_NAME=lstm \  
  icr.io/ibmz/ibmz-accelerated-serving-for-tensorflow:x.x.x
```

8.2.3 Model inferences

You can perform inferences from within the IBM Z Accelerated for TensorFlow container by using one of the following scripts:

- ▶ [credit_card_fraud_rest.py](#)
- ▶ [credit_card_fraud_grpc.py](#).

Both scripts send inference requests to the ModelServer and return the prediction accuracy.

8.2.4 Model profiling

Run the model with the IBM-zDNN-Plugin graph optimizer disabled to establish a performance baseline. Figure 8-2 on page 95 shows a TensorBoard trace view of an LSTM model that is run with the IBM-zDNN-Plugin disabled (**NNPA_DEVICES=0**). The trace provides a baseline performance profile before enabling hardware acceleration.

When you examine the profiling information for a single batch, you can observe that it closely matches the results shown in Chapter 7, “IBM Z Accelerated for TensorFlow” on page 77 when the model runs with TensorFlow.

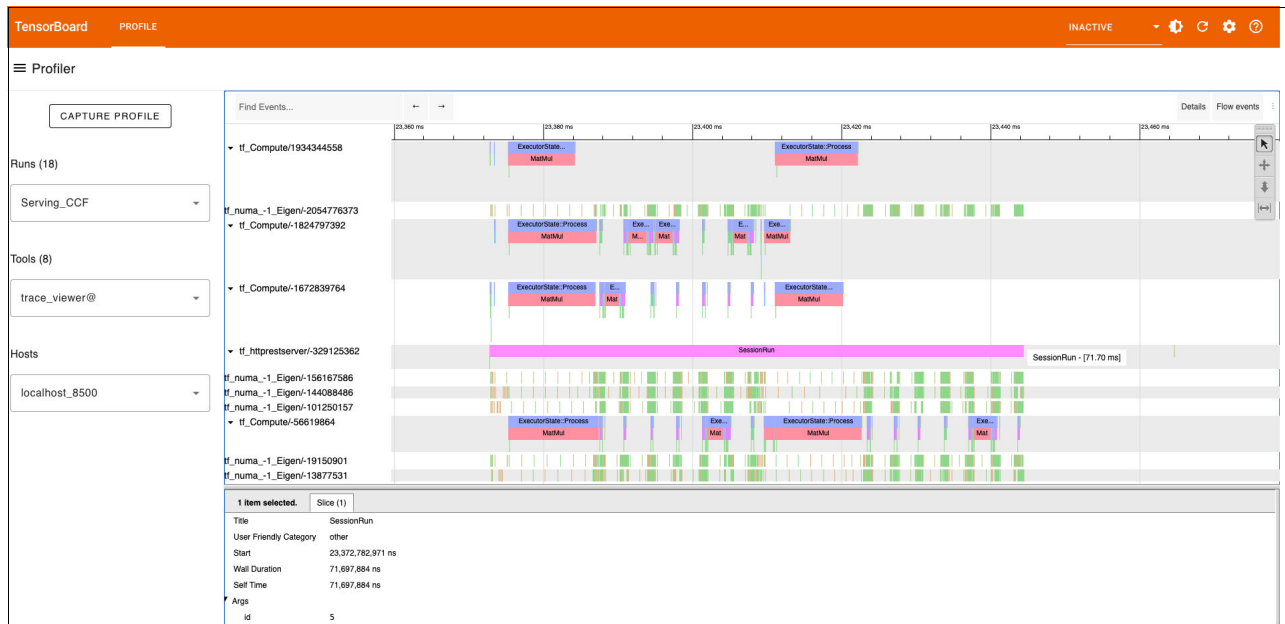


Figure 8-2 LSTM trace view with NNPA_DEVICES=0 using TensorBoard

When you run the use case code with the IBM-zDNN-Plugin graph optimizer enabled, you see a similar view of the captured profiling information. Figure 8-3 shows the TensorBoard Profiler interface displaying a timeline of operations during model execution. The timeline includes events such as SessionRun and ExecutorState::Process, which are represented by colored bars that indicate their duration. The interface also provides detailed metadata for selected operations, including start time, duration, and execution context.

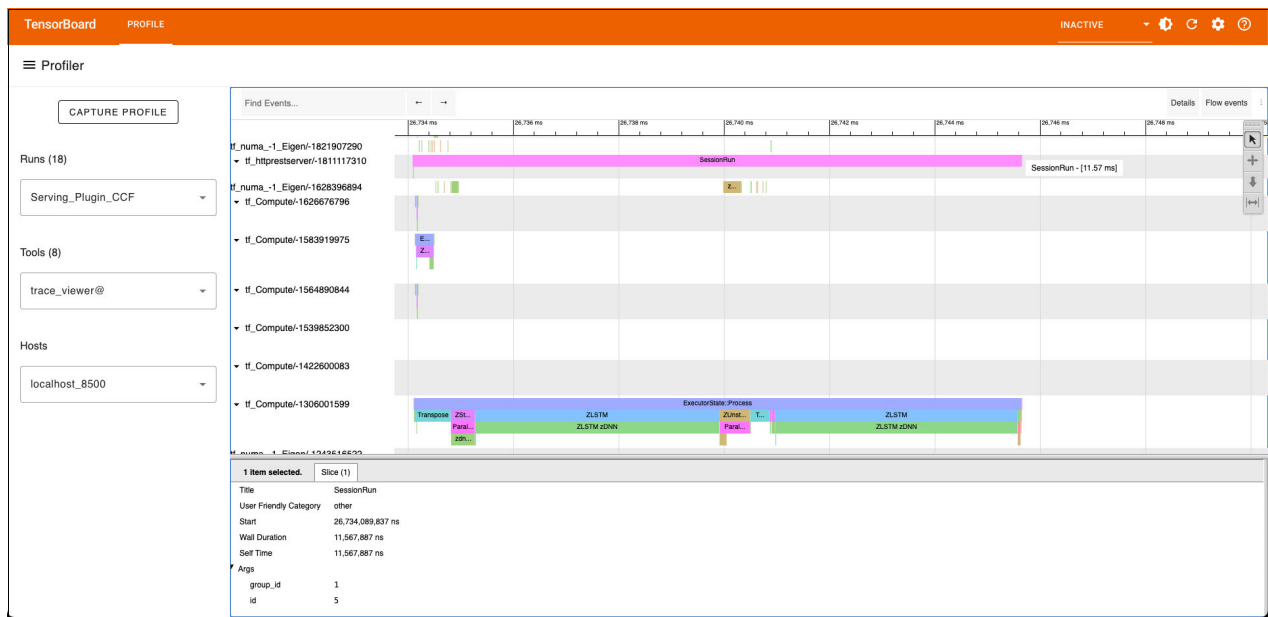


Figure 8-3 LSTM trace view with NNPA_DEVICES=1 using TensorBoard

When you enable the IBM-zDNN-Plugin graph optimizer, each LSTM layer is fused into a single custom operation called ZLSTM. This optimization reduces the runtime for the entire batch from approximately 71.70 ms to 11.57 ms, based on the observed results in this use case.



Other considerations

This chapter focuses on broader aspects and key considerations for deploying and managing the AI Toolkit for IBM Z and IBM LinuxONE. Although these container images provide enhanced machine learning libraries that are tailored for IBM Z, they are not stand-alone solutions. Organizations must implement additional layers of security, infrastructure, and configuration to ensure that these tools comply with internal security protocols and operational policies. This chapter also provides documentation of technical and security guidelines.

In addition to technical considerations, this chapter highlights the importance of understanding and complying with licensing terms when using open-source or third-party code, models, and datasets. Ensure that your usage aligns with both legal and organizational standards. The chapter concludes by outlining the support options that IBM provides, including free access to container images and IBM Elite Support for enterprise users. Links to licensing and support documentation are provided to help you maintain compliance and obtain assistance when needed.

This chapter includes the following sections:

- ▶ 9.1, “Security and technical considerations” on page 98
- ▶ 9.2, “Licenses and support from IBM” on page 98

9.1 Security and technical considerations

The AI Toolkit for IBM Z and IBM LinuxONE container images provide machine learning libraries enhanced for IBM Z. These container images and libraries serve as building blocks for AI and machine learning applications; they are not complete application-hosting solutions.

Often, you must add security components, application software, infrastructure, and configuration to build around these container images. These additions help secure the images from attackers and help ensure that they comply with your organization's security and operational policies.

For a list of general and technical considerations, see [ai-toolkit-for-z-and-linuxone/deployment-guidelines.md](#)

9.2 Licenses and support from IBM

With AI and machine learning technologies, it is common to use open-source or third-party code, models, and datasets both for learning the framework and for building applications. As with any resource that you use from the Internet, ensure that your usage complies with all license terms and follows your enterprise's legal, security, and operational policies.

The AI Toolkit for IBM Z and IBM LinuxONE container images include a combination of open-source and IBM proprietary code. Review the license materials for each product to ensure that they comply with your organization's guidelines. License information is linked in the product documentation for each container image that you use. For more information, see [ai-toolkit-for-z-and-linuxone](#).

The container images can also be used without charge by downloading them from the IBM Container Registry. IBM Elite support is available for AI Toolkit for IBM Z and IBM LinuxONE. More information on support is available from the following locations:

- ▶ [AI Toolkit for IBM Z and LinuxONE](#)
- ▶ [What to Expect From Support](#)

Abbreviations and acronyms

AI	Artificial intelligence
AML	Anti-money laundering
AOT	Ahead-of-Time
BLS	backend lifecycle scheduling
CCFD	Credit Card Fraud Detection
CLI	command-line interface
CSV	comma-separated values
DL	deep learning
DPU	Data Processing Unit
GFP	GraphFeaturePreprocessor
GRU	Gated Recurrent Unit
IBM	International Business Machines Corporation
ICR	IBM Container Registry
IR	intermediate representation
JIT	Just-in-Time
LLM	Large Language Model
LSTM	Long Short-Term Memory
ML	machine learning
MLIR	Multi-Level Intermediate Representation
MLz	Machine Learning for z/OS
NLP	Natural Language Processing
NNPA	Neural Network Processing Assist
OCI	Open Container Initiative
ONNX	Open Neural Network Exchange
PPC1e	PowerPC Little Endian
RF	Random Forests
RNN	Recurrent Neural Network
RNNs	Recurrent Neural Networks
SLAs	service level agreements
TFLOPs	teraflops
TIS	Triton Inference Server
zCX	z/OS Container Extensions
zDLC	Z Deep Learning Compiler
zDNN	Z Deep Neural Network

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *AI for Linux on IBM Z and LinuxONE applications and examples, REDP-5756*

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- PyTorch Documentation

https://docs.pytorch.org/docs/stable/generated/torch.autograd.grad_mode.inference_mode.html

- ONNX MLIR

<https://github.com/IBM/onnxmlir-triton-backend>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



REDP-5749-00

ISBN 0738462330

Printed in U.S.A.

Get connected

